Software Designed
for AMIGA

Lattice C Compiler

Lattice® AmigaDOS C Compiler:

Programmer's Reference Manual

Version 4.00

Lattice, Incorporated
2500 S. Highland Avenue
Lombard, IL 60148
USA

Subsidary of SAS Institute Inc.

Lattice AmigaDOS C Compiler: Programmer's Reference Manual

Copyright © 1987 by Lattice, Incorporated, Lombard, IL, USA.

All rights reserved. Printed in the United States of America. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic or mechanical, photocopying, or otherwise, without the prior written permission of the publisher, Lattice, Incorporated.

Lattice® is a registered trademark of Lattice, Incorporated. dBC III™ and LSE™ are trademarks of Lattice, Incorporated. Amiga™ is a trademark of Commodore Business Machines, Inc. Macintosh™ is a trademark licensed to Apple Computer Inc. IBM® is a registered trademark of International Business Machines Corporation. MS-DOS® and XENIX® are registered trademarks of Microsoft Corp. UNIX® is a registered trademark of AT&T.

Preface

The Lattice AmigaDOS C Compiler is a powerful compiler in the Lattice tradition. This is a reference manual for the Lattice AmigaDOS C Compiler. Its primary purpose is to accurately describe all commands, environment variables, header files and external names that you will normally use while writing and executing C programs.

In order to use the Lattice AmigaDOS C Compiler you need the following minimum requirements:

- 512k of main system memory
- AmigaDOS Version 1.2
- Two 880k 3.5-inch floppy disk drives
- An RGB(A) color display monitor
- A printer device which is supported by AmigaDOS.

This manual consists of nine sections and seven appendices:

- Section 1 describes the steps used to install the compiler and covers the hardware requirements, setting up floppy and hard disk systems, assigning directories and files and logical name assignments.
- Section 2 presents information on AmigaDOS as it impacts the compiler.
- Section 3 covers compiler and linker operation.
- Section 4 is a Lattice C language definition which presents implementation decisions, compiler limitations and new language features within this release.

- Section 5 describes the AmigaDOS C programming environment.
- Section 6 describes using BLINK, the Lattice linker.
- Section 7 contains a list of compiler and linker error messages.
- Section 8 describes the differences between this and previous releases of the compiler.
- Section 9 presents an overview of the Lattice C commands, data function names, file names and environment variables.
- Appendix A is a bibliography.
- Appendix B contains a listing of the files that make up the compiler.
- Appendix C describes the AmigaDOS commands that are provided with the Lattice C Compiler or that must be used in conjunction with the compiler.
- Appendix D describes the public data names that are defined in the Lattice Libraries.
- Appendix E describes the AmigaDOS environment variables that are used by the Lattice C Compiler and related programs.
- Appendix F describes the public function names that are defined in the Lattice libraries.
- Appendix G is a combined reference for all commands, data names, environment variables and function names within the compiler.
- Appendix I is an index.

Contents

Section 1. Installing The Compiler

Hardware Requirements 1-1
Installation 1-3
Setting Up a Floppy Disk System 1-3
Backup with Two Floppy Disk Drives 1-4
Setting Up a Hard Disk System 1-5
Assignment of Directories and Files 1-5
The INCLUDE: Assignment 1-7
The LC: Assignment 1-7
The LIB: Assignment 1-7
The QUAD: Assignment 1-7
Logical Name Assignments 1-8

Section 2. Amiga System Software

The Amiga Operating System 2-1
EXEC System Routines 2-1
ROM Kernel Libraries 2-2

Section 3. Compiler Operation And Use

Using The Compiler And Linker 3-1
Your First C Program 3-2
Using the Math Library 3-4
Math Library Differences 3-5
Direct Calling of Amiga Libraries 3-6
Coding Library Calls for In-line Code Generation 3-7
Built-In Functions 3-10
Header File Compression 3-11

Header File Organization 3-11
Using the FD2PRAGMA Program 3-12
Using Other Libraries 3-13
A Further Programming Example 3-14
Including Other Object Modules 3-15
Advanced Features 3-15

Section 4. Lattice C Language Definition

ANSI C 4-1
Comparison To Kernighan And Ritchie 4-2
Compiler Implementation Decisions 4-9
Pre-Processor Features 4-9
Arithmetic Objects 4-10
Derived Objects 4-12
Storage Classes 4-12
Scope of Identifiers 4-14
Initializers 4-14
Expression Evaluation 4-15
Control Flow 4-17
Compiler Limitations 4-18
New Language Features 4-18
Void 4-18
Enumerations 4-19
Aggregate Assignment 4-20
Passing Aggregates by Value 4-20
Functions Returning Aggregates 4-21
Function Declarations with Argument Types 4-21

Section 5. Programming Environment

Program Sections 5-1
Data Representations 5-3
Sizes and Formats 5-3
Data Portability Issues 5-4
Assembly Language Interfaces 5-6
Entry Conditions 5-7
Register Saving 5-7
Return Values 5-8
Writing Assembly Language Modules For C 5-8
Addressing Modes 5-10
Base Relative Addressing 5-10
PC-Relative Branches 5-12
Short Integers 5-12
Creating a Load-and-Stay-Resident (LSR) Program 5-12

Section 6. Operating The Linker

Using BLINK 6-1
Example Command Line Usage 6-6
Example WITH files 6-6
Introduction To Overlays 6-7
The Overlay Manager 6-9
Using Overlays 6-9
WITH Files 6-10
The MAP File 6-11
Custom Overlay Managers 6-11
Compiler Options and Overlays 6-12
Other BLINK Options and Overlays 6-12
Overlay Examples 6-13
Map Files 6-16
Special Names 6-18
Terminology 6-19

Section 7. Error Messages

Compiler Error Messages 7-1
Operational Errors 7-2
Syntax Errors and Warnings 7-4
Internal Errors 7-14
BLINK Error Messages 7-17

Section 8. Version Differences

Introduction 8-1
Summary of Compiler Changes 8-2
Summary of Library Changes 8-8
Miscellaneous Changes 8-12

Section 9. Compiler Functions

Introduction 9-1
Commands 9-1
Data And Function Names 9-3
Portability Classifications 9-3
Function Calling Sequences 9-4
Argument Type Checking 9-6
File Names 9-7
Environment Variables 9-7

Appendix A. Bibliography

C Programming References A-1
AmigaDOS References A-4
Motorola 68xxx References A-6

Appendix B. Disk Contents

Disk 1 B-1
Disk 2 B-2
Disk 3 B-3

Appendix C. Commands C-1

Appendix D. Data Names D-1

Appendix E. Environment Variables E-1

Appendix F. Function Names F-1

Appendix G. Combined Reference G-1

Appendix I. Index I-1

Section 1. Installing the Compiler

This manual is designed as a reference source for the Lattice AmigaDOS C Compiler. Its primary purpose is to accurately describe the following:

- Commands.
- Environment Variables.
- Header Files.

as well as external names which you will normally use while writing and executing C programs.

If you are already familiar with C and AmigaDOS, then you should have no problems using this manual as a guide to both installing and operating the Lattice C Compiler. However, if you are new to C and/or AmigaDOS, you should begin to familiarize yourself with the necessary background by consultation of some of the publications listed in Appendix A, Bibliography.

Several of these publications should be considered a mandatory requirement if you are to proceed successfully in utilizing C as the vehicle for your programming on the Amiga. The publications concerned are:

- *Amiga ROM Kernel Manual* (two volumes)
- *The C Programming Language*

Full details of these are given in Appendix A together with sufficient information for you to be able to order them from any bookstore.

Hardware Requirements

The compiler cannot be used effectively unless your computer is equipped with at least 512 kilobytes of main memory and two disk drives. This means that users with a *'naked'* A1000 model (viz. an unexpanded 256k unit with a single internal disk drive) will face severe memory problems. Indeed, a 256k A1000 model cannot be considered a suitable vehicle for serious programming using the C language. The A500 and A2000 models do not have this problem of memory shortfall as they are provided with sufficient memory for compiling operations by the manufacturer. However, the A500 and basic A2000 will require an additional drive.

While it is not impossible to operate the compiler on a single drive system, the exercise would be tedious in the extreme due to the numerous disk changes necessary during compilation.

The OPTIMUM system requirements are really constrained by what you can afford to purchase, but a typical system configuration would look something like this:

- 512k of main system memory
- AmigaDOS Version 1.2
- Two 880k 3.5-inch floppy disk drives
- An RGB(A) color display monitor
- A printer device which is supported by AmigaDOS. Refer to your *'User Guide'* supplied with the system for further details of printers.

For *'power users'* and software developers, a PROFESSIONAL system configuration as used by our in-house programmers looks like this:

- 512k of main system memory coupled with 2.0MB of expansion memory giving a total of 2.5MB of memory.
- AmigaDOS Version 1.2
- Two 880k 3.5-inch and one 440k 5.25-inch floppy disk drives
- One hard disk drive with a *minimum* capacity of 20MB.
- An RGB(A) color display
- A dot-matrix printer supported by AmigaDOS and capable of a minimum print speed of 150 cps.

The best results will be obtained with the second configuration, since memory expansion and a hard disk greatly improves performance and vastly simplifies the program development cycle.

One question often asked by aspiring Amiga programmers relates to which device to purchase first – the hard disk or additional memory. Without doubt the expansion of memory should always take precedence over a hard disk. Whilst we are not proposing that you take your system up to its maximum memory expansion of 8.5MB on the A1000 model or 9.5MB on the A500 or A2000 models, an additional 2.0MB seems to satisfy virtually any programming need.

Installation

The Lattice C Compiler package for the Amiga is distributed on three diskettes using the Amiga standard 3.5-inch, double-sided format. The three disks are not copy-protected and are designed to allow for an easy transfer to a hard disk if required.

Before you start using the C Compiler we strongly recommend that you make a working copy of the distribution diskettes. The Lattice Program License Agreement (appearing on the envelope containing the distribution diskettes) allows you to make ONE back-up copy of the program diskette. This copy is for your sole use and should only be used as your working copy. Keep your distribution diskettes in a safe place and utilize the working copy of the C Compiler for your day-to-day operations.

There are two sets of instructions relating to the installation of the C Compiler, one for Floppy Disk Systems and the other for users with a Hard Disk System. Please follow the instructions for the type of system you are using.

Appendix B lists the contents of the diskettes. One file of particular importance is called READ.ME, and is located in the *root* directory. This is a text file containing the latest product information, bug fixes and general notes about the product.

Access to this file can be through the *'Click For Info On Lattice C'* icon or you can use the AmigaDOS *TYPE* command to obtain a listing of this file. Alternatively, you can use an editor such as the Lattice Screen Editor product to view the file.

Setting Up a Floppy Disk System

The installation of a floppy disk system is basically done in two parts. The first element of the process is to make a duplicate of each disk and then to *ASSIGN* the various components of each disk. The latter process is vital to the successful operation of the compiler since, in basic terms, assignment of directories informs the system of the location of these directories.

Making duplicates of the disks is made extremely simple if you are operating under the *'Workbench'* environment. It is done by dragging the icon of each distribution diskette icon onto another disk's icon. If *Workbench* is not already installed, in other words you cannot see disk icons on the right hand side of the screen after resizing your current CLI window, type the following command:

loadwb

and press the RETURN key. This action will place you in the *Workbench* environment where you can see the disk icons.

Backup with Two Floppy Disk Drives

This is considerably easier and less time-consuming than using just a single disk drive system.

Step 1. Ensure that the *write-protect* tabs are set to the protected position on your distribution diskettes. Obtain and label the backup diskettes.

Step 2. Remove any disks currently in the disk drives and place one of your distribution diskettes in the internal (DF0:) drive. Place the disk which is to serve as your backup disk in the external (DF1:) drive. Ensure you are in the *Workbench* environment, (see above for details).

Step 3. Move the mouse pointer to the distribution diskette icon and *drag* this icon over the icon for the backup disk, before *releasing* the icon being dragged.

Step 4. Follow the on-screen prompts for the duplication.

Step 5. When the duplication process has ceased, remove the distribution diskette and rename the backup disks.

Step 6. Remove the backup disk AFTER the disk drive lights have extinguished themselves. Repeat the process until all the disks have been duplicated.

Step 7. Store your distribution diskettes in a cool, dry place. DO NOT leave them lying around your work area.

You should now read the subsection titled *'Assignment of Directories and Files'* which follows the next subsection dealing with hard disk installation.

NOTE: Experienced Amiga users who are familiar with the CLI environment may wish to use the AmigaDOS *'DISKCOPY'* command. This is marginally quicker than using the *Workbench* environment.

Setting Up a Hard Disk System

Physical installation is only necessary when a hard disk system is going to be used. The recommended hard disk installation procedure is to execute the *install_hd* batch file located in the 'S' directory of disk #1. Note that this can only be accomplished from the CLI environment. Several subdirectories will be created during this process. These subdirectories contain the header files and a number of C and assembler source files. You may want to edit this batch file to suit your own requirements in terms of directory locations – remember to adjust the assignments if you have done any re-arrangements.

For example, assume your working floppy disk is titled *'Lattice_C'*, the command:

execute Lattice_C:s/install_hd

will result in the following:

- If necessary, creates the directory SYS:lc
- copying of the executable files to the C: directory
- copying of the header files to the directory named SYS:lc/include
- copying of the libraries and object files to the directory SYS:lc/lib
- copying of the run-time source files to the directory SYS:lc/source.

The batch file *'install_hd'* will not copy the examples from the floppy disk. Some of the source files included are for demonstration purposes, while others contain parts of the Lattice run-time system which you may want to change.

Assignment of Directories and Files

The compiler package is a bootable system. In other words, Disk #1 can be used in place of your normal Workbench disk. If you are using floppy disks you do not need to perform any special installation procedure. Disk #1 is a bootable disk which leaves you in the CLI environment with the necessary logical names already assigned.

When the system prompts you for the Workbench disk during the boot procedure, either from a system power-up or from a *soft* (keyboard) reset sequence, insert the compiler Disk #1 into the internal (DF0:) drive.

The *Workbench* (icon-based environment) is not loaded by this boot procedure. On a 512k Amiga this can be worth around 100k and allows the maximum amount of memory for program development. If you require Workbench, then enter the following AmigaDOS command at the CLI prompt:

loadwb

If you wish to customize your program development environment, Appendix B contains a list of the files which are part of the compiler package. You can locate the files anywhere you wish (on any disk), provided you make the logical name assignments.

The Lattice C Compiler needs to know about four files or logical names which must be assigned to directories before compilation can take place. If you are unsure about *assign*, refer to your reference manuals on AmigaDOS. The names of these assignments are:

- INCLUDE:
- LC:
- LIB:
- QUAD:

Assignments are done using the AmigaDOS *ASSIGN* and *PATH* commands. The PATH variable is used by AmigaDOS version 1.2 when it is loading programs for execution. If the specified program is not in the current directory, then AmigaDOS searches each of the directories mentioned in PATH.

For example, if programs can be found in *DH0:c* and *DH0:lc*, then you should use the following command to establish the *PATH* variable:

PATH DH0:lc ADD

The INCLUDE: Assignment

The INCLUDE: assignment is used by the compiler to locate what are termed *'system header files'*. These are the files which are fetched via the #include operation, thus it is of vital importance that the compiler knows where to find them. This assignment should refer to the directory in which you normally keep the header files.

For example:

- *assign INCLUDE: DH0:lc/include*
- *assign INCLUDE: Lattice_C:include*

The LC: Assignment

The LC: assignment allows the driver program to find the two compiler phases if they are not in the current directory or the *C:* directory.

For example:

- *assign LC: DH0:c*
- *assign LC: Lattice_C:c*

The LIB: Assignment

The LIB: assignment is used to identify the location of the run-time support files and libraries for linking. If this assignment is not made, the driver program *lc* assumes the libraries reside in the directory *SYS:lc/lib*.

For example,

- *assign LIB: Lattice_C:lib*
- *assign LIB: DH0:lib*

The QUAD: Assignment

The driver program uses the QUAD: logical name assignment to allow the default location of the compiler intermediate file to be somewhere other than the source file directory. Placing the quad file on a different drive can improve compile times, although the most significant improvement is realized by creating and maintaining your quad files in RAM.

In fact, the complete compilation cycle will be significantly improved by locating and assigning everything into the RAM: device. This is the principle benefit of expanding the system memory, however, always remember to write your files to disk storage BEFORE executing a finished program.

Logical Name Assignments

Logical name assignments may also be used as a form of shorthand when typing AmigaDOS commands. For instance, when manually invoking the linker, it is much easier to type:

```
LIB:lc.lib,LIB:amiga.lib
```

than it is to type:

```
DH0:lc/libraries/lc.lib,DH0:lc/libraries/amiga.lib
```

These assignments should be added to the file *s/startup-sequence* on the Workbench disk you normally use to bootup from, or the compiler disk #1. The actual assignments you make depend upon the installation method you use. These are as follows:

1. Hard Disk Installation:

   - *assign LC: DH0:c*
   - *assign INCLUDE: DH0:lc/include*
   - *assign LIB: DH0:lc/lib*
   - *assign QUAD: RAM:*

   This assigns *LC:* to directory *c* on the hard disk DH0:, *INCLUDE:* to subdirectory *SYS:lc/include* and *LIB:* to subdirectory *SYS:lc/lib*.

2. Floppy Disk Installation:

   - *assign LC: Lattice_C:c*
   - *assign INCLUDE: Lattice_C:include*
   - *assign LIB: Lattice_C:lib*
   - *assign QUAD: RAM:*

This assigns *LC:* to the *c* directory of the floppy disk volume named *Lattice_C*, *INCLUDE:* to the directory *include* on the same floppy, and *LIB:* to the subdirectory *lib* of the same floppy disk volume.

NOTE: These assignments are made automatically if you boot the compiler disk #1. If you install the compiler in some other way, you should make whatever assignments are appropriate to enable *lc* to find the various files.

Section 2. Amiga System Software

It would be a fair assumption to state that the majority of prospective Amiga programmers have originated from an 8-bit processor background. While some knowledge and experience in 6502 or Z80 assembly language all helps towards an understanding of the 68000 processor contained within the first-generation of Amigas, the concepts of *'multi-tasking'*, *'round-robin scheduling'*, *'task contention'* and *'soft-machine architecture'* are more than likely to be both new and rather mysterious terms.

Those programmers who have C programming experience on mainframes, minicomputers, in a UNIX environment or other proprietary operating systems should have little, if any difficulty in comprehending these concepts. This does not mean that they should skip this section, rather they should view this section as background to the Amiga operating system.

The Amiga Operating System

Programming the Commodore Amiga family using the Lattice C Compiler is made very easy since the bulk of the operating system is also written in C. This means that you can directly access the numerous system routines with a minimum overhead in terms of time and code size.

It is very important to realize that unlike the operating system in 8-bit machines or the IBM-PC, the Amiga operating system does not employ *'hard'* addresses for system routines. In other words, the screen memory does not have a specific location – it can, and is allocated to where the operating system controller (known as *'EXEC'*) decides. This is what the term *'soft-machine architecture'* means, it is *EXEC* making the decision on memory locations and not the programmer. Unless you plan to program in 68xxx assembly language, please forget the concept of memory locations. When programming the Amiga in C, it is both unnecessary and potentially dangerous to the overall stability of the system.

EXEC System Routines

EXEC serves as the core of the Amiga's operating system and removes from the programmer the responsibility of many of the mundane tasks associated with control of the machine. For example, memory allocation is controlled by *EXEC* and in the event where insufficient memory exists *EXEC* will cause the system to issue a visual warning called an *'alert'* to the user. This allows a recovery to be made from what could be a potential system failure or *'crash'*.

Without delving too deeply into the system software, at this stage it would be wise to confine your knowledge of *EXEC* as a sophisticated component of the system software manipulating the clever hardware components such as the 'Blitter' and 'Copper'. *EXEC* could be described as the 'processor interrupt controller' since it controls the actions of the 68000 processor by 'time-slicing'. This means that if we have a situation where there are four tasks for the 68000 processor to carry out, *EXEC* will allow each task a fixed amount of processor time before diverting the 68000 onto the next task.

From a programming viewpoint it is not possible to alter the length of time each 'time-slice' represents, although it is possible to assign a priority to a task. Thus a task could be given a higher priority than any other current task and this would mean that this task receives a greater proportion of processor time and is able to execute faster than the other tasks. The screen display is an example of a task requiring a high priority and the operating system is designed in such a way that the 68000 processor can be forced to wait while the 'Blitter' manipulates the screen display.

EXEC does share something in common with the system software routines and is based on the concept of 'lists'. These could be thought of in terms of 'jump-tables' or 'vector-tables' either leading to specific system routines or holding data details required by *EXEC* to control a task. This data would consist of such things as memory locations of related routines, memory requirements, task priorities, etc. These 'lists' are very important to the overall concept of the system software which is collectively known as the *'ROM Kernel'*.

EXEC lists are highly structured and designed with upward-compatibility in mind. This means that the *'ROM Kernel'* routines are able to operate with other members of the 68000 family such as the 68010 virtual-memory processor or the 68020 32-bit processor. In fact *EXEC* is sophisticated enough to be able to detect which processor the system is running on. This explains why it is possible for a processor such as the 68010 which is physically compatible to the 68000, can directly replace the latter in the A500, A1000 and A2000 models.

ROM Kernel Libraries

As mentioned earlier, the Amiga family is built around a *'soft-machine'* architecture. There is only one absolute memory location in the system which a programmer needs to be aware of, namely location $00000004 which is the address of the pointer to the base, or entry point of *exec.library*. This memory location, ($00000004) never changes irrespective of the Amiga model. Thus, system routines and lists need not always have the same address locations but are placed in memory wherever there happens to be an available location. The two principle advantages to this are:

- Very efficient utilization of available memory in a multi-tasking environment.
- Portability, since with no 'fixed' addresses your code is less dependent on the hardware configuration.

The drawback to this type of architecture revolves around speed as various forms of 'indirect addressing' have to be resorted to. However, it is a function of the compiler to overcome this speed overhead by generating clean, efficient code. In addition to this, there is some hardware assistance to the problem in the form of the 25 DMA (Direct Memory Access) channels within the current generation of Amigas.

If you examine the disk containing the header files you will see that they are organized into named libraries such as the *'graphics library'*, *'exec library'* and *'intuition library'*. These three libraries are probably the most important and are easily the largest in size. Before any library routine can be used, the library has to be *'opened'* by *EXEC* – in case you were wondering who or what is responsible for opening the *'exec library'*, don't worry as this is done automatically for you when you make your first 'open library' call.

The contents of each library is fully detailed in the Commodore-Amiga documentation in the manuals concerned with the ROM Kernel. A brief outline of the contents of the libraries is shown below:

CLIST — The odd-sounding name is an abbreviation for *'character list'* and this library permits you to initialize (prepare) a block of memory prior to carrying out *'string manipulations'*. These manipulations can consist of moving bytes or words to the start or finish of a linked list, conversion of *'clists'* to continuous data blocks, such as null-terminated C strings. This library also permits string concatenation (joining), indexing and length operations.

DISKFONT — This library currently only contains two routines, the first is to load a font into memory, and the second is to construct an array of all fonts available in either memory or on the disk.

DOS — Basic disk input/output and process control is operated from this library through the AmigaDOS functions of *open*, *close*, *read*, etc.

EXEC	This library contains the routines for device and resource management, memory and interrupt control, linked list manipulation, input/output handling, task control together with message and port handling. At a much lower level and with some restrictions, there are routines to directly control the 68000 through the processor status register.
GRAPHICS	By far the largest in terms of the range and number of available functions, this important library handles *RastPorts* (drawing primitives), *ViewPorts* (screen display primitives), *BitMaps* (graphics data areas) and *GELS* (Graphic ELementS). The *GELS* allow access to a further range of components such as *sprites*, *Vsprites*, *Bobs*, *AniObs*, text and fonts. The graphics library permits you to safely access the special graphics hardware of the 'Blitter' and 'Copper'.
ICON	These are routines used by the Workbench to allocate/deallocate memory for Workbench objects and icons. These are items such as 'tools', 'projects', 'drawer', etc., as well as updating icon filenames such as *'copy of Wombat'*, *'copy 2 of Wombat'*.
INTUITION	This library contains routines which are constructed on top of the graphics library, layers library and the console device. INTUITION provides the user with a standardized visual interface irrespective of the application. The elements of this library contains screens, windows, mouse handling, pull-down menus, gadgets, requesters, Preferences and input/output using the IDCMP (Intuition Direct Communications Message Port).
MATHFFP	The mathematical routines of addition and subtraction, multiplication and division, integer conversions, negation and comparisons using the Motorola single-precision *'fast floating point'* conventions are carried out through this library.
MATHIEEEDOUBBAS	This library contains the same routines as the preceding mathffp library except that the format is double-precision

IEEE. In simple terms this means slower code, but with greater accuracy. There is an additional library named *'MATHIEEESINGBAS Library'* which operates to single-precision accuracy.

MATHTRANS — Translation of Motorola fast floating point format to IEEE double-precision format is one of the functions available from this library. There are routines for fast floating point numbers involving trigonometric functions such as *sin, tan, cos, arcsin, arctan, arccos, sinh, tanh, cosh, log, power* and *square root.*

TRANSLATOR — This library currently contains a single routine to convert text strings into phonetic strings.

> NOTE: The libraries listed above are only those which are accessible to C programmers.

The following libraries are not loaded into memory and are held on the disk until required by the system:

- Diskfont Library
- Icon Library
- Mathtrans Library
- Mathieeedoubbas Library
- Translator Library

A very good way to find out and familiarize yourself with the system software is to print out and carefully examine these header files. They contain a great deal of useful information and in many instances have explanatory comments attached. Note that many of these files are extremely large and you will require a lot of paper and several binders!

At this point you should have gained some notion of what lies beneath the surface of these versatile machines. The most important rule in terms of any programming is to always follow the conventions specified by Commodore-Amiga. There is nothing to stop you from ignoring these conventions, shutting down *EXEC* and doing everything yourself by programming the machine as you would program one of those 8086 machines. However,

such an action would immediately remove the multi-tasking facilities and since this is a major feature of the machine, software which does this is likely to be extremely unpopular among users. Not least, this form of programming is completely against the design philosophy of the system software and has serious consequences for code portability in future models.

Section 3. Compiler Operation and Use

The Lattice AmigaDOS C Compiler uses the classical development sequence for creating programs. This sequence looks like this:

- EDIT
- COMPILE
- LINK

In other words, you must first create a source file containing your C program. This is fed to the compiler, which produces an object file. Object files then serve as input to the AmigaDOS linker, which produces an executable file. Your development sequence now look like this:

ACTION	FILE TYPE	FILE EXTENSION
Edit	Source	.c
Compile	Object	.o
Link	Executable	none

Using The Compiler And Linker

Source files are normal AmigaDOS ASCII text files. These consist of lines of text separated by linefeeds and contain standard USA keyboard characters. AmigaDOS contains two editors for operations on files. These are known as *EDIT*, a batch line editor and *ED* a standard text editor. The uses of *EDIT* are rather limited and this editor will not be discussed further.

ED can be used to produce suitable C source files although it should be appreciated that the facilities offered by *ED* are restricted. You would be well advised to purchase a professional text editor such as the Lattice Screen Editor (LSE). This has a large number of advanced features and is extremely popular among many software developers.

Your First C Program

For very simple programs, you can just copy the source lines directly from your keyboard to the source file using the AmigaDOS indirection command. To try this, make sure you have carried out the following actions:

- you are in the CLI mode
- you have installed the Lattice C compiler in the recommended manner
- your *PATH* and logical name assignments were made as suggested in the first section.

Try this example and enter the following and press the RETURN key at the end of each line. Ensure that you copy this exactly in terms of capital letters and spaces. Note that line 5 is done by pressing the CTRL key and then pressing the backslash '\' key while maintaining the pressure on the CTRL key:

```
cd ram:
copy * hello.c
#include <stdio.h>
void main() { printf("Hello World!\n"); }
CTRL-\
lc -L hello
hello
```

If everything works satisfactorily, then you will see the phrase *'Hello World!'* printed on your screen immediately after typing the last line.

Now let's examine what you did. The first line:

```
cd ram:
```

This is a standard AmigaDOS command to place the *'current directory'* in RAM: and speed up operations by avoiding the use of a disk drive.

The second line:

```
copy * hello.c
```

This is another AmigaDOS command which begins copying lines from the console (i.e. your keyboard) to a file named *'hello.c'*.

Assuming this is your first effort, then the file does not exist in RAM: and AmigaDOS creates this file for you. If the file already exists from a previous attempt at compiling, then AmigaDOS will overwrite the old file.

```
#include <stdio.h>
  void main()     { printf("Hello World!\n"); }
```

The next two lines are pure C language and should be familiar to you. If you are new to C or do not fully understand these lines, refer to Section A, Bibliography for a list of recommended books on C programming.

Each line you have typed will now be directly written to the file until you enter a line consisting of a *CTRL-* character. This character signifies to AmigaDOS a break in the character stream and input is then curtailed.

Line six which looks like this:

```
lc -L hello
```

invokes the both phases of the compiler to compile and *'link'* your first program named *'hello.c'*. Following this command, you should see some Lattice copyright messages on your screen.

You should not see any error or warning messages appear on your screen during the compilation phases. If this does occur, then there is obviously something wrong somewhere. Check carefully the following:

- Examine your source program *'hello.c'* for spelling mistakes.
- Make sure you have set the assignments *PATH* and/or *INCLUDE* correctly.

If your first program was compiled successfully, *LC* invokes the linker, specifying the relevant input and library files. The process of *'linking'* means in simple terms, the binding of your program code with that of the compiler. During the execution of the linking phase you should see some copyright information, although there should be no error or warning messages.

The final line containing the command *'hello'* executes the compiled program which has been named *'hello'* by the linker. This program simply prints the phrase *'Hello World!'* to the screen and then terminates.

If you were unable to complete these steps with satisfactory results, you should consult your C tutorial material or go over the complete process with a friend who has knowledge of AmigaDOS and/or Lattice C experience. If you live in the USA then you can contact the Lattice Technical Support department. Please note that while we are willing to assist, we cannot function as a remote tutor for beginners to the C language. In many cases we have found that many first-time users often run into problems at this point and waste a great deal of time because of simple errors.

Using the Math Library

When you are comfortable with the *'Hello World!'* program, the next step is to attempt compiling and executing some of the demonstration programs supplied on your copy of your distribution diskettes. Note the use of the word *copy*, make sure you are not using the original diskettes. The demonstration source programs are normally installed in the *'source directory'*.

For example, the source file named *'ftoc.c'* converts temperatures from Fahrenheit to Celsius and requires you to link with the math library. Observant readers will notice that this program is based on the temperature conversion program listed on page 8 of the Kernighan and Ritchie reference work. You can run this program using the following commands:

```
cd LC:source
lc -Lm ftoc
ftoc
```

Once again, you must install the compiler in the standard way and make the appropriate logical name assignments if this program is to compile and link successfully.

Note that we specified the inclusion of the Lattice math library file called *'lcm.lib'*. The reason for this is whenever you use the *'double'* or *'float'* data types in your program or when you call any of the functions defined in the *'math.h'* or *'float.h'* header files. Make a careful study of these files in order to gain an understanding of how and why the compiler uses header files.

Math Library Differences

There are actually three math libraries supplied with the Lattice AmigaDOS C Compiler. These are named:

- LCM.LIB
- LCMFFP.LIB
- LCMIEEE.LIB

A description of each library follows:

LCM.LIB — The standard math library *LCM.LIB* contains all the mathematical functions documented within this manual. This library directly supports the IEEE format for representing floating point numbers.

LCMFFP.LIB — The math library *LCMFFP.LIB* offers support for the Motorola Fast Floating Point (FFP) library as supplied by Commodore-Amiga. This library also contains a subset of the math routines within the standard math library. The FFP library should be used whenever you compile a program with the fast floating point option *-f*. This format and the routines available from the FFP library are documented in the Amiga ROM Kernel manuals.

LCMIEEE.LIB — The third math library *LCMIEEE.LIB* provides a linkage to IEEE format routines supplied by Commodore-Amiga on the Workbench diskette. This library contains low level routines such as add, subtract, multiply, and divide that can be used in place of the same routines in the standard math library. Note that this library does not contain any transcendental functions. If this library is specified, you must additionally specify the standard math library.

The reason why there are three math libraries is related to numerical accuracy. For applications demanding the highest accuracy such as a financial accounting package or possibly a spreadsheet then the IEEE math library is probably the best one to use. The overhead to using this library is speed since this library is slower in operation than the FFP library. Thus, if your application is primarily based on speed rather than accuracy then the FFP library is probably your best option. Try a compilation sequence using both libraries to make a comparison.

When you manually link a Lattice C program, the standard Lattice run time support library *LC.lib* should always be placed just before the Amiga run time support library *amiga.lib* in your command line. If your program uses floats or doubles, the appropriate math library should be specified prior to the standard library.

For example, in order to link *ftoc* for the FFP format you would use the following command line:

blink lib:c.o,ftoc.o to ftoc lib lib:lcmffp.lib,lib:lc.lib,lib:amiga.lib

Direct Calling of Amiga Libraries

In general, the bulk of an Amiga C program will consist of various library calls. Improvements to this version of the compiler mean that library routines do not have to go through the stub routines in Amiga.Lib, although the option remains for those who wish to still do this. This gives the advantage of suppressing the code to push the function parameters on to the stack, calling the stub routine which places them into the appropriate register and then calls the library. The compiler can be informed which registers to place the parameters into and this allows it to select the optimal method of register loading.

There are two advantages to avoiding the use of the Amiga.Lib stub routines:

- Code execution is faster than code using stub routines.
- Development time is saved by the avoidance of linking.

The code generated by this technique does not usually show any significant saving in size over code generated by the conventional Amiga.Lib route. However, the stub routines can account for a large proportion of a small program and the execution speed advantage can become important.

Calling of these direct functions is done through the inclusion of the *prototype* header files. These are located on Compiler Disk #2 with the calling syntax of:

```
#include <proto/lib-name.h>
```

To use the *Open()* function of AmigaDOS, the calling sequence would look like this:

```
#include <proto/dos.h>
```

Coding Library Calls for In-line Code Generation

It should be realized that the techniques used for the library function calls are not hard coded for the known Amiga functions. This means that these function calls may be freely used in any source code, although it is necessary to code a statement of the form:

#pragma libcall <base> <routine> <offset> <magic>

where:

#pragma Is part of the special calling routine.

libcall Is part of the special calling routine.

<base> Is the pointer to the library base.

<routine> Is the name of the library routine you intend to directly call.

<offset> Is the offset from the library base of the routine.

<magic> Is the description of the parameters. These are described later in this section.

An example call using the *Open()* function of AmigaDOS would take this form:

#pragma libcall DOSBase Open 1e 2102

where:

DOSBASE Is the library pointer.

Open Is the name of the function being called.

1e Is the offset from the library base as a hexadecimal value.

2102 Is the value used by the compiler to determine the register conventions.

Note that the offsets from the library base actually have negative values, this is ignored in the interests of clarity.

The calculation of the *magic* portion is arrived at breaking down the number into nibbles representing the register conventions. The format of the *magic* number is:

4321r#

where the numbers 4, 3, 2, and 1 indicate the registers where the fourth, third, second and first parameters respectively, are to be placed. The values to use for these are:

HEX VALUE	DATA REGISTER	HEX VALUE	ADDRESS REGISTER
0	D0	8	A0
1	D1	9	A1
2	D2	A	A2
3	D3	B	A3
4	D4	C	A4
5	D5	D	A5
6	D6	E	A6
7	D7	F	A7

Note that address registers A5 to A7 are never used in the above context. Observant programmers may recognize this pattern as that assigned by Motorola to the registers in the processor instruction set.

The *'r'* portion indicates the register where the result is returned in. This is usually 0 for data register D0, although there are no restrictions as to which register you may select for return values.

The *'#'* value indicates the total number of parameters for passing to the routine. The high-order bit (0X8) of this nibble indicates that the routine to be called unreasonably destroys the contents of data registers D6 and D7. In most cases you may ignore this nibble completely since it is intended to provide a Kickstart version 1.1 compatibility with the *Text()* and *DoIO()* functions.

If we return to our example function call:

#pragma libcall DOSBase Open le 2102

and examine the *magic* and offset values, you will see how these figures were arrived at by an examination of vol.2 of the *Amiga ROM Kernel Reference Manual* page A-122. The synopsis of the *Open()* function is:

```
file = Open( name, accessMode )
D0           D1      D2
```

Reference to the previous table will show that data registers D2, D1 and D0 have the values of 2, 1 and 0 respectively, and this forms the first part of the value 2102.

If you examine the function *OpenLibrary()* on page A-124 in vol.2 of the *Amiga ROM Kernel Manual*, the synopsis is:

```
library = OpenLibrary(libname, version)
 D0                    A1       D0
```

The first part of our *magic* value would then be 090, the second part *parameter* value is 2, thus the final calculated value is 0902.

The offset values can be obtained from the v1.2 .fd files on the v1.2 *Extras* disk. Alternatively, the columns starting on page D-5 of vol.1 of the *Amiga ROM Kernel Manual* provide these. For our example, the offset value is shown on page D-6 in exec.lib and has a value of 228.

Note that the current edition of vol.1 of the *Amiga ROM Kernel Manual* has several errors in the .fd files listing. One of the most readily identifiable errors is the showing of the incorrect bias for the DOS library – add the decimal value 30 or hexadecimal value 1E to the respective columns. There are also several exec.lib functions missing from the end of the exec.lib listing. A good policy is to always check the .fd files on your disks rather than rely solely on the reference manuals.

Built-In Functions

This version of the compiler includes several built-in functions designed to generate high quality 68000 code. These functions are encouraged by the ANSI standard over others which might have previously been used. The built-in functions are:

- *strlen*
- *strcmp*
- *strcpy*
- *memset*
- *memcmp*
- *memcpy*

The compiler will automatically recognize a built-in function provided it is prefixed by the string:

*_builtin_*

The *#defines* in the header files *string.h* and *stdlib.h* have been modified to accomplish the invocation of a built-in function. To take advantage of these functions it is necessary to include the correct header file – for these functions it is *string.h* file. In situations where you do not want to use the built-in functions, you can modify the appropriate function with an *#undef*.

An additional built-in function has been provided to assist in the speedier development of small programs. This is:

printf

When this function is called, the compiler examines the formatting string:

- When it is a constant string with no substitutions, the compiler changes the *printf* call to a *_write* call.
- When it is a constant string AND only contains *%s, %d* and *p%* formats a call is made to *_tinyprintf*.

The purpose behind this is to allow programmers to use *printf* for output without having to write a trimmed version which does not handle the unused formats. Depending on the type of feedback we receive, we may add others such as *scanf, sprintf, fprintf*, etc.

The advantage of built-in functions is related to the ability of the compiler to exploit register contents. For example, the *strlen* call will generate no code if the input string is a constant string. In addition, functions cannot return the result pointer if it is unused – all of this tends to a regime which produces significantly less code than an equivalent call to the library function.

Header File Compression

The utility *lcompact* can be used to compress header or source files in a form that can be read much faster by the compiler. The utility operates by encoding commonly occurring Amiga keywords. Although ANSI is quite specific about the processing of header or include files we believe this is a method of increasing the overall compiler performance without sacrificing ANSI compliance.

To use the utility, you need to feed the source file from *stdin* and send the result to *stdout*. The command line syntax looks like this:

lcompact <source>dest

We have supplied the header files in both compressed and uncompressed forms. There is no difference in the handling of these – you do not have to specify anything special on your command lines or in the *#include* statement. You may also have compressed and uncompressed header files within the same directory.

Header File Organization

There is now a new subdirectory with the header files called *proto*. This subdirectory contains a number of header files which contain function prototypes and the magic material which allow the Amiga Library functions to operate. If you are calling an Amiga function, you should include the appropriate header file. However, if you want to lazy, you can use the syntax:

```
#include <proto/all.h>
```

This will include all the header files although an increase in compilation time will be the result. Where the system configuration contains expanded memory and everything is being held in memory, the increase in compilation time is probably academic.

Note that these files correspond directly to the *.fd* files since they were constructed from them.

Using the FD2PRAGMA Program

The format of the *#pragma* statements for library calls is relatively easy to understand and code, it is difficult to maintain and or decode. We have provided a utility which will take a standard ASCII *.fd* file and convert this file to the appropriate *#pragma* statements.

The *.fd* file should consist of single statements per line and structured in the following form:

- A line to be processed as a directive starts with two (hash) symbols.
- An asterisk (*) at the start of a line symbolizes a comment line.
- Any other line indicates entry points into the library.

The directives for the conversion utility are:

base _name	Identifies the global variable pointing to the base of the library.
bias *n*	Identifies the offset value between the library base and the initial vector in the library.
public	Identifies subsequent lines as being available to programmers.
private	Identifies subsequent lines as being available to the library only.
end	Marks the end of the *.fd* file.

The entry points into the library take the form:

entryname(arguments)(registers)

with commas separating the individual arguments and registers. The latter may also be separated by a slash character '/'. Some example entries look like this:

InitCode(startClass,version)(D0/D1)
PrintIText(rp,itext,left,top)(A0/A1,D0/D1)

The first example is from exec.library while the second example is from intuition.library – note the mixing of commas and slashes on the register descriptions.

We strongly suggest that you always maintain and update the *.fd* file, running it through the FD2PRAGMA utility rather than trying amend the #pragmas themselves. This technique is less likely to lead to errors.

Conversion of the *.fd* file is done by a command line of the following syntax:

fd2pragma alpha.fd beta.h

where:

alpha.fd Is the *.fd* file for conversion.

beta.h Is the new header file after conversion.

Any errors encountered in the conversion will be printed on the standard output file. Typical errors would be invalid directives, invalid register specifications or no *end* statement.

Using Other Libraries

There are many useful libraries available for the Lattice AmigaDOS C Compiler. For example, the Lattice MacLibrary for portable applications between the Amiga and Apple MACINTOSH families, or the Lattice dBCIII Library for database applications using the popular MS-DOS application. If you live in the USA contact Lattice directly for full details about these and other Amiga products. Alternatively ask your Lattice authorized local distributor about prices and availability.

The library documentation for these Lattice products explains how to link with Lattice object modules, and all products include linking batch files. Ensure that libraries purchased from other sources include this information.

The basic rule for using other libraries is to always mention them before the Lattice math and standard libraries. If you use the driver program *LC*, you may specify additional libraries immediately after the *-L* option. Refer to the section dealing with commands for a complete description of the *LC* command.

A Further Programming Example

This is a program which uses one of the system libraries, in this case the Intuition library. The function of this program is to rapidly 'flash' the display screen 10 times. Rather than use direct input from the keyboard, you will be creating the program as a file using *ED* or the Lattice Screen Editor. Refer to the relevant documentation for creating a file and name it *flash.c* or a name of your own choice, provided a '.c' file extension is used. Make certain that the code is entered EXACTLY as it is shown below.

Here is the source code to be entered:

```
/*   Screen Flash and Beep using Intuition    */

#include <exec/types.h>

extern APTR OpenLibrary();
extern VOID CloseLibrary(),DisplayBeep();

APTR IntuitionBase

   main();
   {
      int i;

      IntuitionBase = OpenLibrary("intuition.library',0);

      if (IntuitionBase != 0)  {

         for (i = 0;i  < 10;i++)  {
               DisplayBeep(0);
               Delay(3);
         }

        CloseLibrary(IntuitionBase);
      }
   }
```

Assuming you have successfully saved the file to disk, the next step is to copy the file to RAM: to save a little time. The commands are:

```
copy df0:flash.c ram:
cd ram:
```

This copies your file from the directory of drive df0: to RAM: and places you in the directory of RAM: and ready to go. Make any necessary changes to these commands where there are variations on file name or drive. Compile the file in the same way as your first C program named *hello* using the command line:

```
lc -L flash
```

All that is necessary after a successful compilation is to enter the name of the file and your screen should 'flash' a yellow color very quickly. If you do receive error message from the compiler during compilation, check your source file for mistakes. A good place to start is by checking the semi-colons ';' since leaving these out or putting them in the wrong place is a common error by beginners.

Including Other Object Modules

In many situations you may want to include more than one object module during the linking process. For example, if your application consists of two object modules named *'ALPHA.O'* and *'BETA.O'* and you want the executable program to be named *'GAMMA'*, then you could invoke the linker with the following command:

```
blink lib:c.o,ALPHA.O,BETA.O to GAMMA lib lib:lcmffp.lib,lib:lc.lib,lib:amiga.lib
```

Advanced Features

Once you are comfortable building simple C programs, you will probably want to familiarize yourself with the more advanced features of the compiler. These features include different addressing modes and special compilation options. This information is presented in the section containing the detailed specifications for the compiler and linker commands.

Never be afraid to experiment, this is the best way to learn about both the Amiga, compiler and the C programming language. A very good way of familiarizing yourself with the styles and conventions of Amiga programming is to examine the source code available on the huge number of *public domain* software disks. Your local distributor, dealer or User Group should be of assistance here. Always remember – every single Amiga programmer started with a zero knowledge level of the machine, so there is plenty of help available out there – all you have to do is ask.

Section 4. Lattice C Language Definition

This section describes the C language as supported by Version 4.0 of the Lattice compiler. The Lattice AmigaDOS C Compiler product conforms very closely to the compilers associated with UNIX System V and ANSI C. In most instances the differences, particularly with the former system are trivial and for the most part are a restriction of the hardware rather than the software.

ANSI C

The ANSI committee for C standardization has chosen to use the UNIX V definition as their starting point for this standard, and the Lattice compiler conforms to the proposed ANSI standard. The committee has also proposed several new additions to the C language, and the active involvement of Lattice Inc. means your Lattice C compiler will always conform to the very latest international standard provided you continue within the update process.

The most important of these new additions, a feature called *'argument type checking'*, has been incorporated into the Lattice compiler. The others will be incorporated as they become more clearly defined.

Please note that we are not providing a complete specification of the C language in this part of the manual. Appendix 1 cites a number of books which can perform this function. In addition to the mandatory Kernighan and Ritchie we recommend that you obtain one of the more recent C programming books which discuss the latest UNIX and ANSI features of the language.

In this section, we will assume that you have a good general knowledge of the C language obtained from such books and some previous experience with another C compiler.

We will then describe the peculiarities of the Lattice compiler, which fall into three categories:

- How the C language as supported by Lattice differs from the specification provided in Kernighan and Ritchie? We use Kernighan and Ritchie as the basis for comparison as the documentation describing the proposed ANSI C standard is not yet readily available

- How does Lattice C handle the murky, undefined areas of the language which are usually written off by most compiler vendors as *'implementation issues'*?
- What are the processing limits of the Lattice compiler?

In addition, if you have been using an earlier version of the Lattice C compiler, you will find Version 4.0 to be considerably different, both in language features and in library features.

Comparison To Kernighan And Ritchie

The most precise definition of the C programming language generally available is found in Appendix A of the Kernighan and Ritchie text, which is entitled *'The C Programming Language'*. The following list highlights the differences between this version of Lattice C and the Kernighan and Ritchie definition. You may want to annotate your copy of Kernighan and Ritchie to take account of the differences.

The items use the Kernighan Ritchie numbering scheme. For example:

CRM 2.1 Comments

refers to the paragraph marked 2.1 in the Kernighan and Ritchie text.

CRM 2.1 Comments

The default mode is that comments do not nest. A compile time option can be used to allow nested comments. Here is an example of nested comments:

```
/*  Begin comment block #1
.
.
/*  Begin comment block #2 (This is
nested within block #1)
.
.
*/  End comment block #2
.
.
*/  End comment block #1
```

CRM 2.3 Keywords

Lattice C includes two additional keywords, *void* and *enum*. These are ANSI-compatible.

CRM 2.4.1 Integer Constants

Names declared as values for an enumeration type may be used as integer constants.

CRM 2.4.3 Character Constants

Two new escape sequences are recognized:

- \v Specifies a vertical tab (VT) character.
- \x Introduces one or two hexadecimal digits which define the value of a single character. For example, '\xf9' generates a character with the value 0xF9.

By default the compiler only permits a single character to be defined, a compile time option can be used to permit multiple character constants. The result will be a short integer for a two-character constant and a long integer for three or four characters.

CRM 2.5 Strings

The same \x convention described above can be employed in character strings, where it is generally more useful. Additionally, a compile time option can be used to force the compiler to recognize identical string constants and only generate one copy of the string. Note that when a quoted string is used to initialize a character array, the string is not actually treated as a string constant, since it is actually placed into the array at compile time.

For example,

```
char abc[] = "1234";
char *p = "1234";
```

will produce a 5-byte array named *'abc'* containing the digit characters from 1 to 4, followed by a null

byte. The second declaration produces a string constant similar to the contents of *'abc'* and places a pointer to the constant into *'p'*.

CRM 2.6 Hardware Characteristics — Refer to Section 5 of this reference manual.

CRM 4.0 What's In A Name? — Each enumeration is conceptually a separate type with its own set of named values. The properties of an enum type are identical to those of int type. The void type is used when declaring a function which has no return value.

CRM 6.0 Conversions — An expression can be converted to the void type by means of a cast. This is often used to explicitly indicate the discarding of a function return value. However, an expression of type void cannot itself be converted or used in any way.

CRM 7.1 Primary Expressions — The Lattice compiler always enforces the rules for the use of structures and unions in order to determine the intended set of member names. Since the compiler maintains a separate set of member names for each structure or union, the primary expression preceding a . (period) or the -> operator must be immediately recognizable as a structure or as a pointer to a structure of the type which contains the specified member name.

For example, given the following declarations:

```
struct A
{
  int x;
  int y;
} m,*p;

struct B
{
  int xx;
  int yy;
} n,*q;
```

The following statements would be invalid:

```
p = &n;  /* p must point to structure
type A */
q = &m;  /* q must point to structure
type B */
```

You can convince the compiler that you only really want to generate code for these statements by using the appropriate cast operations. For example:

```
p = (struct A *)(&n);
q = (struct B *)(&m);
```

CRM 7.2 Unary Operators

The requirement that the & operator can only be applied to an lvalue is relaxed slightly to allow application to an array name (which is not considered an lvalue). Note that the meaning of such a construct is a pointer to the array itself, which is quite different from a pointer to the first element of the array.

The difference between a pointer to an array and to an array's first element is only important when the pointer is used in an expression with an integral offset, because the offset must be scaled (multiplied) by the size of the object to which the pointer points. When the pointer points to the array instead of to the first array element, the target object size is the size of the whole array, rather than the size of a single element.

CRM 7.7 Equality Operators

The only integer to which a pointer may be compared is the integer constant zero.

CRM 7.14 Assignment Operators

Both operands of the simple assignment operator (=) may be structures or unions of the same type.

CRM 8.1 Storage Class-Specifiers

The Kernighan and Ritchie text states that the storage class-specifier, if omitted from a declaration outside a function, is taken to be *extern*. This is somewhat misleading, if not plainly inaccurate. In fact as the Kernighan and Ritchie text points out in CRM 11.2, the presence or absence of *extern* is critical to determining whether an object is being defined or referenced.

- If *extern* is present, then the declared object either exists in some other file or is defined later in the same file. But if no storage class specifier is present, then the declared object is being defined and will be visible in other files.
- If the static specifier is present, the object is also defined but is not made externally visible.

The only exception to these rules occurs for functions, where it is the presence of a defining statement body that determines whether the function is being defined.

The Lattice compiler can be forced to assume *extern* for all declarations outside a function by means of the -x compile time option. Declarations which explicitly specify static or extern are not affected.

CRM 8.2 Type Specifiers

Two new type specifiers are supported, as mentioned earlier:

- void
- enum

Also, the compiler recognizes the following new types which are specified via multiple keywords:

- unsigned short
- unsigned short int
- unsigned long
- unsigned long int

CRM 8.5 Structure and Union Declarations

The Lattice compiler maintains a separate list of member names for each structure and union. A member name may not appear twice in a particular structure or union, but the same name may be used in several different structures or unions within the same scope.

Structure and union tags are in the same class, this means that you cannot use the same tag for both a structure and a union.

Enumerations are declared in much the same way as structures and unions, with a separate list of identifiers for each enumeration type. Enumerations are unique types which can only assume values from a list of named constants. The C language treats them as *int* values but restricts operations on them to assignment and comparison. The named constants may appear wherever an *int* is legal.

The optional name which may follow the keyword *enum* plays the same role as the structure or union tag; it names a particular enumeration. All such names share the same space as structure and union tags.

The names of enumerators in the same scope must be distinct from each other and from those of ordinary variables.

CRM 8.7 Type Names

Although a structure or union may appear in a type name specifier, it must refer to an already known tag. In other words, structure definitions cannot be made inside a type name. Thus, the sequence:

```
(struct { int high, low; } *) x
```

is not permitted, but:

```
struct HL { int high, low; };
   (struct HL *) x
```

is acceptable.

CRM 10.2 External Data Definitions

The Lattice compiler applies a simple rule to external data declarations. If the keyword extern is present, the compiler assumes that the actual storage will be allocated elsewhere, and the declaration is a reference to the storage area which will be resolved by the linker. Otherwise, the declaration is interpreted as an actual definition which allocates storage, unless the -x option has been used, as described previously under the CRM 8.1 heading.

CRM 12.3 Conditional Compilation

The constant expression following #if may not contain the *sizeof* operator and must appear on a single input line.

CRM 12.4 Line Control

Although the filename for #line must be an identifier, it need not conform to the characteristics of C identifiers. The compiler takes whatever string of characters is supplied, and the only syntax requirement for the filename is that it cannot contain any white space.

CRM 14.1 Structures and Unions

Structures and unions may be assigned, passed as arguments to functions, and returned by functions.

The escape from typing rules as described in Kernighan and Ritchie is explicitly not allowed by the Lattice compiler. In a reference to a structure or union member, the name on the right must be a member of the aggregate named or pointed to by the expression on the left. Our implementation, however, does not enforce any restrictions on references to union members, such as requiring a value to be assigned to a particular member before allowing it to be examined via that member.

Compiler Implementation Decisions

This section describes how the Lattice compiler deals with some aspects of the C language which have fallen into the domain of *'implementation decisions'*. Some of these issues were left up to the compiler designers simply through oversight in the language definition, and the ANSI committee is gradually closing these loopholes. Others cannot be resolved in the language definition because they depend upon the specific hardware or software limitations faced by the compiler designers.

Pre-Processor Features

The Lattice C compiler supports the full set of pre-processor commands described in Kernighan and Ritchie. Pre-processor commands are handled concurrently with lexical and syntactic analysis of the source file, because there is no requirement to have a separate pre-processor pass, and compiler performance is improved by this approach.

Nevertheless, analysis of the pre-processor commands is largely independent of the compiler's C language analysis. For example, #define text substitutions are not performed for pre-processor commands, but nesting of macro definitions is possible because substituted text is re-scanned for new #define symbols.

Since the compiler uses a text buffer of fixed size, a particularly complex macro may occasionally cause a line buffer overflow condition. Usually, however, this error occurs when there is more than one macro reference in the same source line, and it can be circumvented by placing the macros on different lines.

Circular definitions such as:

```
#define A B
#define B A
```

will be detected by the compiler if either A or B is ever used, as will more subtle loops.

Like many other implementations of C, the Lattice compiler pushes macro definitions onto a stack, so that if the line:

```
#define XYZ 12
```

is followed later by:

```
#define XYZ 43
```

the new definition takes effect, but the old one is not forgotten. In other words, after encountering:

```
#undef XYZ
```

the former definition (12) is restored. To completely undefine XYZ, an additional #undef is required. The rule is that each #define must be matched by a corresponding #undef before the symbol is truly forgotten. Also, the compiler issues a warning message whenever a macro is re-defined. This was done to help detect those nasty little errors which occur when header files are in conflict.

Two clarifications should be noted with regard to the #if command.

- An undefined symbol in a #if expression is treated as having a value of zero.
- A symbol defined with null substitution text is interpreted as having a value of one.

These conventions are consistent with #ifdef usage, and permit the use of expressions like:

```
#if SYM1 ¦ SYM2 ¦ SYM3
```

which causes subsequent code to be processed if any of the symbols are defined.

Arithmetic Objects

Five types of arithmetic objects are supported by the Lattice compiler. These, as well as pointers, are the basic data entities which can be manipulated by a C program. The types are:

- short (also, short int)
- char
- long (also, long int)
- float
- double (also, long float)

In addition, the keyword unsigned may be applied as a modifier to any of the integral data types. This modification does not affect the size of the object, and is significant only when such an object is used in an expression involving:

- conversion
- multiplication
- division
- comparison
- bitwise right shifts.

One very important point to consider in terms of portability of your code is the integer size on the Amiga. The natural size of an integer in the AmigaDOS environment is 32 bits.

This means the types *int* and *long* are equivalent. However, if you want your program to be fully portable to other environments, use *short* when you really want a 16-bit integer, and use *int* when you want a general integer for such things as loop counters. Note that this equivalence of *long* and *int* types is only applicable to implementations for 68xxx-based machines.

In expressions involving several data types, the compiler generally carries out the computation in the *'widest'* data type which is involved. The conversion rules are:

EXPRESSION	COMPUTATIONAL
char	char
short	short
long	long
float	double
double	double

For example, if an expression contains items having types *char* and *long*, the *char* types are converted to *long* during the computation.

It is interesting to note that all floating point computations are done in double precision, which is the traditional C standard. This implies that your program will generally execute slower if you use the *float* data type.

Many programmers mistakenly believe that *float* data is handled faster than *double*. The ANSI C proposal now allows for computation to be done in single precision floating point if no *double* data items are involved, and we will probably incorporate that feature in a later release.

Derived Objects

The Lattice C compiler supports the standard extensions leading to various kinds of derived objects, including pointers, functions, arrays, and structures and unions. Declarations of these types may be arbitrarily complex, although not all declarations result in a legal object.

For example, arrays of functions or functions returning arrays are illegal. The compiler checks for such illegal declarations and also verifies that structures or unions do not contain instances of themselves.

Objects which are declared as arrays cannot have an array length of zero, unless they are formal parameters or are declared as extern.

Note that the size of aggregates (arrays and structures) may be affected by alignment requirements.

Storage Classes

Declared objects are assigned by the compiler to storage offsets which are relative to one of several different storage bases. The assigned storage base depends on the explicit storage class specified in the declaration, or on the context of the declaration, as follows:

External — An object is classified as external if the extern keyword is present in its declaration, and it is not later declared outside the body of any function without the extern keyword.

Storage is not allocated for external items because they are assumed to exist in some other file, and must be included during the linking process that builds a set of object modules into a load module.

Static — An object is classified as static if the static keyword is present in its declaration or if it is declared outside the body of any function without an explicit storage class specifier.

Storage is allocated for static items in the data section of the object module. All such locations are initialized to zero unless an initializer expression is included in the declaration.

When static items are declared outside the body of any function without the static keyword being visible in other files, then they are termed as being externally defined. Note that string constants are allocated as unnamed static *char* arrays.

Automatic — An object is classified as automatic if the auto keyword is present in its declaration or if it is declared inside the body of any function without an explicit storage class specifier. It is illegal to declare an auto object outside the body of a function.

Storage is allocated for auto items on the stack during the execution of the function in which they are defined.

Formal — An object is classified as formal if it is a formal parameter to one of the functions in the source file. Storage is allocated for formal items on the stack by the program which calls the function.

Note that the first phase of the compiler makes no assumption about the validity of the register storage class declarator. Items which are declared register are so flagged, but storage is allocated for them anyway against either the automatic or the formal storage base.

A register variable declaration may be accepted for any pointer or other data object with a size of no more than 4 bytes. Up to two pointers may be assigned to address registers starting with A3 to A2; up to six simple data elements may be assigned to data registers starting with D7 to D2.

The registers are assigned in the same order in which they appear in the function declaration, with formal parameters being assigned first. Any additional register variable declarations are accepted by the compiler without being assigned to registers.

The use of register variables affects the entry sequence at the start of the function in which they are declared, by requiring an additional instruction to save the previous registers' values before they are used in the function.

Note also that if the -x compile-time option is used, the implicit storage class for items declared outside the body of any function changes from static to extern. This allows a single header file to be used for all external data definitions. When the main function is compiled,

the -x option is not used, and so the various objects are defined and made externally visible. When the other functions are compiled the -x option causes the same declarations to be interpreted as references to objects defined elsewhere.

Scope of Identifiers

The Lattice compiler conforms almost exactly to the scope rules discussed in Appendix A of the Kernighan and Ritchie text (pp. 205-206). The only exception arises in connection with structure and union member names, where, in accordance with later versions of the language, the compiler keeps separate lists of member names for each structure or union. This means that additional classes of non-conflicting identifiers occur for the various structures and unions.

Some other points are worth clarifying:

- The compiler does not generate specific allocate/de-allocate instructions for auto items declared in statement blocks within a function. The function entry sequence allocates stack storage to handle the largest collection of declared automatic data items.

- With this scheme, a function may allocate more stack space than is actually used, but the need for run time dynamic allocation within the function is avoided.
- When an identifier with a previous declaration is redefined locally as an *extern*, the previous definition is superseded, but the compiler also verifies compatibility with all preceding *extern* definitions of the same name.
- This is done in accordance with the standards, which require that all references to the same external name must be to the same object.
- The point in this example where a local block redefines an identifier as *extern*, the local declaration does not disappear upon termination of the block as the compiler now has an additional external item for which it must verify later declarations.

Initializers

Objects which are of the static storage class are guaranteed to contain binary zeros when the program begins execution, unless an initializer expression is used to define a different initial value. The Lattice compiler supports the full range of initializer expressions described in Kernighan and Ritchie, but restricts the initialization of pointers somewhat.

A pointer initialization expression must evaluate to the integer constant zero or to a pointer expression of exactly the same type as the pointer being initialized.

This pointer expression can include the address of a previously declared static or external object, plus or minus an integer constant. However, it cannot contain a cast operator applied to a variable, because such conversions cannot be done at compile time.

This restriction makes it impossible to initialize a pointer to an array unless the & operator is allowed to be used on an array name, because the array name without the preceding & is automatically converted to a pointer to the first element of the array. Accordingly, the Lattice compiler accepts the & operator on an array name so that declarations such as:

```
int a[5],  (*pa)[5]  =  &a;
```

can be made. Note that if a pointer to a structure (or union) is being initialized, the structure name used to generate an address must also be preceded by the & operator.

An arithmetic object may be initialized with an expression which evaluates to an arithmetic constant which, if not of the appropriate type, is converted to that of the target object.

More complex objects (arrays and structures) may be initialized by bracketed, comma-separated lists of initializer expressions, with each expression corresponding to an arithmetic or pointer element of the aggregate. A closing brace can be used to terminate the list early. See Appendix A of Kernighan and Ritchie for some examples.

Unions may also be initialized like structures. The initialization list must correspond to the first member of the union.

A character array may be initialized with a string constant which need not be enclosed in braces. This is the only exception to the rule requiring braces around the list of initializers for an aggregate.

Initializer expressions for auto objects can only be applied to simple arithmetic or pointer types (not to aggregates), and are entirely equivalent to assignment statements.

Expression Evaluation

All of the standard operators are supported by the Lattice compiler, using the standard order of precedence as described on Kernighan and Ritchie page 49. Expressions are evaluated using an operator precedence parsing technique which reduces complex expressions to a sequence of unary and binary operations involving at most two operands.

Operations involving only constant operands (including floating point constants) are evaluated by the compiler immediately, but no special effort is made to re-order operands in order to group constants. Thus, expressions such as:

```
c - 'A' + 'a'
```

must be parenthesized so that the compiler can evaluate the constant part:

```
c + ('a' - 'A')
```

If at least one operand in a binary operation is not constant, the intermediate expression result is assigned to a temporary storage location. The temporary then replaces the binary operation in the expression and becomes an operand of another binary or unary operation. This process continues until the entire expression has been evaluated.

The use of temporaries is optimized by the compiler so as to minimize regeneration of identical temporaries during a straight-line code sequence. Thus, common sub-expressions are recognized and evaluated only once.

For example, in the statement:

```
a[i+1] = b[i+1];
```

the expression i+1 will be evaluated once and used for both subscripting operations. This same optimization strategy eliminates expressions producing useless results with no other side effects, such as:

```
i+j;
```

Three conditions cause temporaries to *'die'* during a straight-line code sequence:

- A function call may have side effects which cause previously-computed temporaries to no longer be accurate, and so all temporaries are discarded after the call.
- If either operand associated with a temporary is the result of a later operation, the temporary is discarded after that operation.
- When the result of an operation is stored through a pointer, the compiler discards all temporaries constructed from operands having the same type as the pointer.

The third condition is brought about because the compiler cannot determine if the pointer refers to a component of a current temporary value, and so it takes the safe approach.

Note that this strategy may fail if the programmer uses *'type punning'*. For example, if variable A is declared as an integer and you construct a character pointer which points to A's storage location, the compiler will not discard temporaries containing A when you change A via that pointer.

Except for common sub-expression detection, which may replace an operation with a temporary value, expressions are evaluated in left-to-right order unless this is prevented by operator precedence or parentheses.

However, it is best not to make any assumptions about the order of evaluation, since the language definition allows C compilers to re-order expressions in order to generate better code.

The language definition does guarantee that the logical OR (||) and the logical AND (&&) operators will be evaluated in left-to-right order.

Control Flow

The C language offers a variety of statement flow constructs, and the Lattice compiler supports the full complement of them. Since control flow operations tend to dominate many programs, the compiler takes special steps to optimize them, as follows:

- Switch statements are analyzed to verify that they contain:
    - At least one case or default entry.
    - No duplicate case values.
    - Not more than one default entry.
- The compiler then chooses the best of several machine code sequences to test for the various cases.
- The size and number of branch instructions is kept to a minimum by extensive analysis of the flow within each function.
- Unreachable code is discarded, with the appropriate warning messages.

Compiler Limitations

The more important internal limits of the compiler are listed below:

- The maximum value of the constant expression defining the size of a single subscript of an array is two less than the largest unsigned target machine *int*. These values are:
    - 65,533 for a 16-bit *int*
    - 4,294,967,293 for a 32-bit *int*
- The maximum length of an input source line is 512 bytes.
- The maximum size of a string constant is 256 bytes.
- Macros with arguments are limited to a maximum number of 16 arguments.
- The maximum length of the substitution text for a #define macro is 512 bytes.
- The maximum level of #include file nesting is 10.

It is very unlikely that any of these limitations will have a detrimental effect on AmigaDOS C programs in terms of compiler performance. However, we felt that release of this information would be of value to those programmers whose applications would be taking the hardware to its design limits

New Language Features

This section provides simple examples of the use of several of the new language features which were not described in the Kernighan and Ritchie text.

Void

The reserved word *void* is used to describe a function which has no return value:

```
void noval();                /* function with no return value */
void (*vp)();                /* pointer to void function */
```

While *void* operands may never be used in an expression, it is useful to cast something to *void*:

```
int xfunc();                 /* function returning int */
(void)(xfunc(a, b));          /* discard return value */
```

The use of the cast shows that the programmer is intentionally discarding the return value from *xfunc*.

Enumerations

An enumerated data object is integral but may only assume values from a specified list of identifiers, which can be viewed as integer constants. The actual values assigned to the identifiers normally begin at zero and are incremented by one for each successive identifier in the list.

An explicit value can be forced for any identifier by using an = assignment. Subsequent identifiers are assigned that new value plus one, and so on. Here is an example:

```
/* defining an enumerated data type */

      enum color {red, blue, green = 4, puce, lavender};

/* defining enumerated data objects and pointers */

      enum color x, *px;

/* using enumerated data objects */

       x = red;
       *px = x;
       if (x == lavender) px = &x;
```

In this example, the symbols associated with the enumerated data type color are assigned the following values:

0 = > red
1 = > blue
4 = > green
5 = > puce
6 = > lavender

Aggregate Assignment

Structures or unions of identical type may be copied by assignment:

```
struct XYZ
{
  int x;
  double y;
  long z;
} x, y, *px;

x = y;          /* copies contents of y to x       */
*px = x;        /* copies x to the pointed to struct */
```

For purposes of assignment or passing to functions (see below), structures of identical type may also appear in a conditional expression:

```
y = (i > 30) ? x : *px;
```

Passing Aggregates by Value

A structure or union which appears in an argument list without a preceding ampersand (&) is passed by value to the function:

```
struct XYZ q;

functn(q);
```

The called function must make an appropriate declaration:

```
void functn(a);

struct XYZ a;
```

Because many existing programs pass the address of a structure without using an ampersand (&) operator, the compiler generates a warning whenever an aggregate is passed by value to a function.

Note that the Lattice implementation of aggregate passing by value actually supplies the called function with a pointer which it uses to make a local copy of the caller's structure immediately upon entry.

Functions Returning Aggregates

A function may return an entire structure or union as a return value:

```
struct XYZ fxyz(a, b, c)
int a;
double b;
long c;
{
  struct XYZ r;

    r.x = a;
    r.y = b;
    r.z = c;
  return (r);
}
```

The return value must be copied by assignment to another aggregate of the same type:

```
extern struct XYZ x, fxyz();

x = fxyz(2, 20., 2000L);
```

The Lattice implementation actually returns a pointer to a static copy of the returned aggregate. Because this copy persists only long enough to assign the return value, such functions are still recursively re-entrant (but not, in general, re-entrant in a multi-tasking context).

Function Declarations with Argument Types

External functions may be declared with an enclosed list of type names corresponding to expected arguments. When the function is called, the compiler checks arguments against the types specified in the prototype.

Warning messages are issued if the actual argument expressions supplied to the function do not agree with the expected type or number of arguments. If additional arguments may be present for which no type checking is desired, the list of type names may be ended with a comma.

Consider the following examples:

```
extern char *sbrk(int);
extern FILE *fopen(char *, char *);
extern double sin(double);
extern void fprintf(FILE *, char *, );
```

Programmers should note that three different warnings may be generated when calls to a function declared with argument types are made.

- If the number of arguments does not agree with those present in the prototype, warning 87 is issued.
- If an expression defining an argument fails to agree in type with its corresponding declared type name, warning 88 is issued but the expression is not converted to the indicated type.
- If the expression is a constant, warning 89 is issued and the value is converted accordingly.

Section 5. Programming Environment

Programming with the Lattice AmigaDOS C Compiler is not particularly different from programming in C on a UNIX system. This is an important statement, because it implies that you can use almost any of the popular C programming books with the Lattice compiler. Appendix 1 lists some of the publications which you may want to consider.

If you confine your programming to the features which most of these books describe, you do not need to know a great deal about the Amiga environment. However, if you will be doing special things such as linking with assembly-language functions, then you will need to know more about the low-level aspects of the compiler and run time libraries.

The following topics are most important:

- Program Sections
- Data Representations
- Assembly-Language Interfaces

Program Sections

The compiler isolates you from the details of the 68000 processor instruction set, but you still need to know a little about how your C program is translated into executable code. The object modules produced by the compiler consist of several sections, called *'hunks'* by the AmigaDOS reference manuals. These modules are made up of the following types:

CODE — This contains the machine instructions to execute your program operations. The Lattice compiler produces no self-modifying code sequences, so this section can be placed into read-only memory (ROM). The section is named *'text'* if the -s option is used when the module is compiled; otherwise it is unnamed.

DATA — This contains all initialized data items, including literals and constants. This section cannot be placed into ROM because it may contain writable items. The section is named *'data'* if the -s option is used when the module is compiled; otherwise it i s unnamed.

BSS — This contains all uninitialized data items, which are writable and cannot be placed into ROM. The section is named *'udata'* if the -s option is used when the module is compiled; otherwise it is unnamed. Note that items in this section do not occupy an y space in either the object module or the load module; the memory is allocated by the operating system and cleared to zeroes when the program is loaded.

There are two other program sections which are not part of the object file but which form an important part of the final executing program.

The first of these is the stack, and its size is established by the AmigaDOS *STACK* command. The default size is 4000 bytes if no stack command is used to change the value. The stack is used for context saving during function calls, for parameter passing, and for all automatic storage.

Although many C programs use only a few automatic variables, others may require substantially more than 4000 bytes. We recommend that the stack size be increased to at least 20K for running the compiler. You may find it necessary to use even larger values for certain programs, depending on:

- The amounts of automatic storage used in functions.
- The depth of nesting of functions calls.

Manipulation of the stack value is very much a matter of experience. The technique of changing the value should be approached from an *increasing size* direction. In other words, *increase* the stack size until a satisfactory value is achieved – do not operate the compiler by starting with a stack size of say, 400k and working *downwards*! It should be remembered that the stack size is a finite resource derived from *chip* memory. This means that the more stack your application claims for its own use, implies less memory available for other applications – a point worthy of serious consideration on a multi-tasking machine.

The second additional section is the *HEAP*, also called the memory pool. This is a variable-sized area containing *'free memory'* which you can allocate and de-allocate via library functions.

The amount of memory available for the heap depends upon two factors:

- Total system memory size.
- Memory already in use by the operating system and other active tasks.

The library functions make calls to the operating system to obtain this memory. The requested size is passed directly to the operating system if *sbrk* is used, but if *getmem* is used, the size of the heap is increased by rounding up the requested size according to the value in the external integer *_mstep*.

This approach avoids the high overhead cost of the operating system processing many small allocation requests of AmigaDOS. If your program primarily makes calls to *getmem* for large blocks, a large value for *_mstep* may severly fragment existing memory i.e. leave large *'holes'* in the memory pool. In this event, you can define *_mstep* in your main program with a smaller value.

In summary, the Lattice AmigaDOS environment consists of five distinct memory sections:

- code
- initialized data
- uninitialized data
- stack
- heap.

These normally are scattered throughout the system memory, depending on what memory is available when the program is loaded. If heap space is needed, the memory allocators will request memory from AmigaDOS.

Data Representations

The compiler maintains data in forms which can be efficiently manipulated by the 68000 processor. Specifically, multi-byte data is stored with the high-order byte in the lowest address, floating point numbers are kept in the IEEE standard format and pointers contain 32-bit addresses.

Sizes and Formats

The following table gives the sizes of the C data types for operating under AmigaDOS.

TYPE	BITS	Min RANGE	Max RANGE
signed char	8	-128	+127
unsigned char	8	0	255
signed short int	16	-32,768	+32,767
unsigned short int	16	0	65535
signed int	32	-2,147,483,648	+2,147,483,647
unsigned int	32	0	4,294,987,295
signed long int	32	-2,147,483,648	+2,147,483,647
unsigned long int	32	0	4,294,987,295
float (IEEE format)	32	Small: +/-10E-37	Large: +/-10E+38
double (IEEE format)	64	Small: +/-10E-307	Large: +/-10E+308
pointer	32	Low: 0	High: 0xFFFFFFFF

The more complex C data types, such as arrays and structures, are built up from these basic types. On the 68000, all objects larger than a byte must be accessed from an even address, so structures with single character variables or odd length character arrays may be fragmented i.e. include *'holes'* – unnamed bytes which are inaccessible but necessary for word alignment.

These alignment gaps may also be present between elements of an array of structures, if the size of the structure is an odd number of bytes, and between data elements in the same storage class.

Data Portability Issues

On some processors, such as the 8088, performance is unaffected by address alignment. On others such as the 68020 or 8086, while alignment is optional, performance may be improved if special alignment rules are applied to certain of the basic data types.

The Motorola 68000 however, requires the alignment and an attempt to access a 16-bit or 32-bit quantity from an odd address causes a hardware addressing exception.

The Motorola 68010 virtual memory processor is a special case since it employs a different stack frame format from that of the 68000. This means that the 68010 can recover from an address error exception caused by accessing a word located on an odd address. However, your code must provide an exception-handler which can operate upon the extended data held within the 68010 stack frame.

You should be cautious when exchanging data files between an address segmention orientated regime such as the 8086 and a linear address processor such as the 68000.

For example, consider this short program:

```
#include <stdio.h>
main()
{
FILE *fp;
struct
  {
    char x;
    short y;
     } record;

       fp = fopen("testfile","wb");
       record.x = 3;
       record.y = 4;
       fwrite(&record,sizeof(record),1,fp);
       fclose(fp);
       }
```

If you compile and execute this program under AmigaDOS, the file named *'testfile'* will contain four bytes in the following order:

```
03  00  00  04
```

Then if you compile and execute it on a typical 8086 system such as an IBM-PC, the file will contain three bytes in this order:

```
03  04  00
```

This may be puzzling for programmers who are new to the 68000 processor but the reasons for this are:

- It is standard practice on the 68000 to align short integers on even byte addresses.
- The 68000 normally outputs integers starting from the low-order byte, while the 8086 starts with the high-order byte.

This all goes to prove that portable programs do not necessarily produce portable data files.

Where you are concerned about data file portability, you will save yourself a lot of grief if you consider the problem during the design of file formats. The easiest way to handle this is to only write ASCII text files, since text file processing is usually fully portable. If you cannot do this, then you will need to write a special translation program to re-align arrays and structures and to rearrange the bytes in integers.

These programs are known in UNIX circles as *'data swabbers'*, and they are nasty to write and awkward to maintain.

One last comment on data portability – pointers are almost completely non-portable and you should almost always avoid writing them to files. It is generally true that pointers saved to a file are meaningless even when read back into the same computer system by a different program.

The reason for this is due to the data being pointed at will not necessarily be in the same place under different execution conditions.

Assembly Language Interfaces

As mentioned earlier, it is not unknown to develop a program entirely in C and then performance-tune it by re-writing certain functions in assembly language. Also, there are some 68000 operations, such as direct access of special CPU and I/O registers, which are not easily handled in C.

This *massaging* of the program is only really worth considering on sections of code which are time-critical. Not only is it extremely time-consuming and largely debatable in terms of diminishing returns there is also the question of breaking some dependencies which may not be immediately apparent.

Entry Conditions

Upon entry to a function, the stack will contain all the function arguments immediately above the return address, which is 4 bytes long. The arguments appear in left-to-right order – the leftmost argument is immediately above the return address in the stack. All arguments are passed by value.

For example:

```
int i;
double d;

func(i,d);
```

This will first push the value of d on the stack (8 bytes) and will then push the value of i (4 bytes) and will then call func. Note that all arguments of type char or short are promoted to int before they are supplied to the called function. However, if you employ the complier *-w* option then shorts and chars will be pushed on to the stack as a word (2 bytes).

Register Saving

The compiler currently assumes that functions preserve all of the following registers:

- D2 – D7
- A2 – A7

Registers A5 and A7 (the stack pointer) must be preserved since they specify the stack frame being used by the calling program.

Register A4 must be preserved because it is expected to contain the data section base value. The other registers in this list must be preserved because they may be in use by the calling program as register variables.

Registers D0 and D1 are used for function return values and do not need to be preserved. Registers A0 and A1 are not used for register variables, and the compiler does not assume that their contents are preserved.

Return Values

Function return values are passed back in one or more registers, depending on the data type declared for the function. The conventions are:

char	The value is returned in D0.B (low byte of D0).
short int	The value is returned in D0.W (low word of D0).
long int	The value is returned in D0.
float	The value is returned in D0.
double	The value is returned in D0 and D1, with the high-order part in D0.
pointer	The value is returned in D0.

Note that functions returning aggregates (structures or unions) actually return a pointer to a static copy of the aggregate. Because this copy persists only long enough to assign the return value, such functions are recursively reentrant but not multi-tasking reentrant.

Writing Assembly Language Modules For C

You can write assembly language modules for inclusion in C programs if you are sufficiently experienced with the 68000 instruction set and if you observe the following:

1. The statements defining the functions should be placed in the text section by preceding them with:

   CSECT *text*

   This is not an absolute requirement, since the functions will be accessible regardless of the section in which they are defined, but it assures them of placement with the C functions during linking (if the -s option is used). Each function entry must be declared in an XDEF statement:

```
            XDEF        AFUNC
            .
            .
            .
AFUNC       EQU         *
```

2. If the module is to define data locations to be accessed (using extern declarations) in C modules, those definitions should be placed in the data section by preceding them with:

   CSECT *data*

   Each data element must be declared in an XDEF statement in order to be accessible in the C modules:

```
            XDEF        D0,D1,D2
D0          DC.L        $4000
D1          DC.W        $8000
D2          DC.L        D0
```

3. Any of the registers D2-D7 or A2-A6 must be preserved by the module, and the return value loaded into the appropriate data registers.

   To call a C function from an assembly language module, you must include an XREF declaration for the function. Before calling the function (via JSR), you must supply any expected arguments in the proper order. After control returns from the called function, the stack pointer must be adjusted to account for pushed arguments.

```
XREF        cfunc
MOVE.L      DO,-(A7)        ;push argument
MOVE.L      D1,-(A7)
JSR         cfunc           ;call function
ADDQ        #8,A7           ;restore stack ptr
```

Data elements defined in a C module may be accessed via XREF statements, as well:

```
XREF        XD2,XD3
.
.
.
MOVE.L      XD2,D0
```

Addressing Modes

The compiler provides options which allow variations in the way data and functions are referenced. These options can be used to reduce load module sizes or improve efficiency. Although there can be disadvantages, significant improvements can often be achieved through judicious use of the appropriate options.

One of the more notable effects of these options is the elimination of a large amount of relocation information from the load module. This can be very significant for programs with a large number of small functions.

Base Relative Addressing

The -b option directs the compiler to reference static data as a 16-bit offset from the base register A4 instead of the normal 32-bit direct reference.

This form of addressing has the advantage of requiring only 4 bytes per instruction instead of the 6 bytes necessary for a direct reference. These instructions also generally execute faster.

However, if the -b option, or base-relative addressing mode, of the compiler is used, it is necessary to combine some or all of the data sections.

Since this addressing mode references data locations as offsets from the contents of A4, this means that the data must reside in a contiguous block instead of the normal scattered distribution. This could be a problem where excessive fragmentation of the memory has occurred prior to loading the data. The solution is to try to ensure that this data is loaded early – preferably after a system reboot, when the memory is *'clean'*. The real answer, as we all know, is for programmers to correctly de-allocate memory after use. This would

avoid wasted memory, corrupted memory-free lists, system alerts and all the other evidence of fourth division programming.

The offset field of the instruction is only 16-bits and this limits the amount of data which can be referenced by base-relative addressing to 64K.

The compiler automatically names the data and uninitialized data sections of modules compiled with the -b option as _MERGED, overriding any names for these sections which may have been specified with the -s flag. The linker will automatically merge sections with this name into the data section referenced via register A4.

It is possible to mix modules compiled with and without the -b flag. As long as modules compiled with the -b flag do not reference any data in modules compiled without the -b flag, no special linking procedures are necessary. Otherwise, it is necessary to use the smalldata option of the linker to cause all data to be combined.

There is one important exception to this. The phase 2 option -c can conflict with the -b option. If the -c option has been used to specify that the data sections are to be loaded into *chip* memory (i.e. the lower 512k of the processor address space), the linker will not merge those sections into the special A4 addressable data section.

Therefore, data compiled with the phase 2 option -c should never be referenced by modules compiled with the -b option. If the -b option is used with the phase 2 option -c, the compiler will display a warning and revert to the normal 32-bit addressing mode during code generation.

If a program compiled with -b will be used as an interrupt handler or with the *AddTask()* function, you must compile with the -y option in order for the program to automatically load A4 with the linker supplied value for the base of the data section.

This is necessary because such a task will not be performing any of the initialization normally done by the startup routine c.o, and therefore would not have a correct value in A4.

PC-Relative Branches

Normally the compiler generates 32-bit function references when a program calls an externally defined function. The -r option will cause all function calls to be PC-relative, or relative to the current location counter.

These instructions are 4 bytes long as opposed to the 6 bytes required for a direct reference. They also execute faster.

However, to be effective the target location, i.e. the function being called, must be within +/-32K of the branch instruction. If the target location is beyond this range, either because it is in another segment or the current segment is too large, BLINK must adjust the instruction to branch through a linker generated jump table. This can increase the amount of time required to perform the branch.

Short Integers

Compiling with the -w option will generate code to pass shorts and integers as 2 bytes on to the stack. This has the benefit of reduced code size which theoretically leads to improved execution speed. However, there remains a catch – a situation with mismatched types will result result in a hardware error since the 68000 is being asked to access an odd aligned, rather than an evenly aligned address. Naturally, this will result in the system falling off the end of the world together with a swift visit from the *Guru* bearing the meditation number of 00000003, or as a variation, the meditation number of 00000004, depending on how mangled your code is.

The defense against these problems is use of prototypes for your code. Selection of this course of action will provide you with error messages from the compiler, rather than from the Guru. The other primary advantage of employing prototypes is that the compiler will automatically correct the type of parameter being passed.

Creating a Load-and-Stay-Resident (LSR) Program

This version of the compiler provides an alternative startup module named *CBack.o* which allows the programmer to create Load-and-Stay-Resident (*LSR*) code. We strongly recommend that you use *CBack.o* for ANY program in the following categories:

- No requirement for the original CLI window for the purposes of output.
- A program which needs to be memory resident (such as *POPCLI* or *MEMWATCH*).

The design specification of *CBack.o* means that the programmer only has to make minimal amendments to existing code in order to take advantage of *LSR* code.

Linking with *CBack.o* requires the addition of some definitions to your main module and an awareness of the setting up which *CBack.o* carries out. The first step is the addition of some global variables necessary for *CBack.o* to determine how your task is to run:

LONG _stack	Stack space for the created task
CHAR *_procname	Name of the process to be created
LONG _priority	The priority to create the process at
LONG _BackGroundIO	The flag to indicate an I/O requirement

Most *LSR* programs will probably only need a stack size of 4000 bytes (the system default), although this figure of 4000 may need to be experimented with depending on the *LSR* application. There are no restrictions related to the name, although common sense dictates that a name which is easily recalled is probably best. The task priority should be set to 0, once more a baseline figure which may need to be varied. The *_BackGroundIO* flag is special flag required when you wish to perform any type of I/O to the originating console. When this flag is initialized to 0, your code is expected to carry out its own I/O to a separate window – or no I/O as the case may be. The latter is also true for any program which opens it's own window.

The initializing of *_BackGroundIO* to a value other than 0 informs *CBack* to open the default console and pass it to you in the global variable *_Backstdout*. It is the programmers responsibility to close this file.

> NOTE: Failure to close this file will result in the original window from which the program was run NEVER closing. Also, until you close the file handle, the window will remain open.

While this may be useful for printing out logon banners, diagnostic messages or results of commands, a window which cannot be closed under any circumstances other than a system reboot is indicative of a low standard of programming.

_Backstdout is an AmigaDOS file handle and consequently all I/O to this file handle must

be performed with the *Write()* function – NOT the Lattice functions of *write* or *fprintf*. Closure must be only be done with *Close()*. Where a program establishes a *resident* task to communicate with, the file handle should be passed to this task for it to print messages on the CLI window.

When the *_Backstdout* file handle is described as a NULL, your program cannot perform any output to the window. This can occur when the program was invoked from the Workbench environment or from another program which did not have a standard output window available at the time.

The following example demonstrates a *do-nothing* program which uses *CBack* to install itself:

```
#include <exec/types.h>
#include <libraries/dos.h>
#include <proto/dos.h>
#include <string.h>

LONG _stack = 4000;                    /* a reasonable amount of stack
space */
CHAR *_procname = "my_program";
LONG _priority = 0;                    /* run at standard priority */
LONG _BackGroundIO = 1;                /* perform background I/O */
extern BPTR _Backstdout;               /* file handle we will write to
with */

#define MSG "Hello World!"

void main(argc, argv)
int argc;
char *argv[];

{
   if (argc  == 0)        /* called from Workbench - can't do anything
*/
      _exit(0L);

   if (_Backstdout)     /* called from CLI - print out the string */
     {
     Write(_Backstdout, MSG, strlen(MSG));
     Close(_Backstdout);
     }
}
```

Where an *LSR* program operates as a single task for the entire system (such as *POPCLI* or *MEMWATCH*, it should check for the existance of a global message port. If a global message port does not exist, then the *LSR* program should create one. After creation of the port, any control information should be directed through this port. Where a global message port already exists then this should serve as the primary port for control information. The sample code for *POPCLI* and *MEMWATCH* demonstrate how this can be achieved.

Section 6. Operating the Linker

This section refers to BLINK version 7.0, a linker provided as a replacement for the original linker supplied for AmigaDOS named ALINK. In addition to being considerably faster in operation than the latter, BLINK also provides many options which are not available with ALINK. There are no changes to command lines and the *WITH* files.

Using BLINK

BLINK is operated by keyword parameters to indicate the selected options. The basic command line syntax is:

BLINK [FROM][ROOT] *beta* [TO *alpha*][WITH *alpha*][VER *alpha*]...
[LIBRARY | LIB *beta*][MAP *alpha* map_options][XREF *alpha*][options]

where:

alpha means a single file.

beta means zero or more file names separated by a comma, plus sign or space.

BLINK provides a large number of keywords to give the programmer a wide number of options. Although some of these may seem superfluous, the intention is to provide the programmer with as many options as possible, even if some of these options appear rather obscure. The following keywords are recognized by BLINK:

ADDSYM This causes BLINK to emit HUNK_SYMBOL records for all symbols in the input object files regardless of whether or not the input object file was compiled with the *-d* option.

This is extremely useful for use with a symbolic debugger such as *Metascope*. In general you should also specify the NODEBUG option with ADDSYM to cause BLINK to ignore any symbol records already in the object modules. This prevents having duplicate symbol definitions in the final executable.

BATCH — This causes BLINK to supply the default value of 0 for all undefined symbols. Normally, BLINK will pause after each undefined symbol to give you an opportunity to correct the error. If you specify the BATCH option, it will not pause.

BUFSIZE *n* — Sets the I/O buffer size for BLINK. By default, all I/O is done in 488 character blocks. However, you can alter the I/O performance by adjusting this number. In general the default is sufficient, but may be decreased if memory is severely constrained (at the price of performance). Note that one buffer is allocated for each file BLINK has open at any one time (a maximum of 4).

CHIP — Specifies that all hunks are to be placed in Chip or display memory regardless of the input object hunk specifications.

DEF *symbol=symbol* — This defines a symbol to be used in the linking process. This is particularly useful in conjunction with the PRELINK option to force certain routines to be pulled from the library even though no references to them exists.

DEFINE *symbol=val* — This defines a symbol to be used in the linking process. This is particularly useful in conjunction with the PRELINK option to force certain routines to be pulled from the library even though no references to them exists.

FANCY *n* — Enables usage of printer control characters in the map file.

FAST — Specifies that all hunks are to be placed in Fast or expansion memory regardless of the input object hunk specifications.

FASTER — A do-nothing option that is included only for ALINK compatibility.

FROM *files* — Specifies the object files that are the primary input to the linker. These object files will always be copied to the root of the object module. You must specify at least one object file for the root. If it appears as the first option to BLINK then the FROM keyword is optional and may be omitted. ROOT is an acceptable synonym for

	FROM. FROM may be used more than once with the files for each FROM adding to the list of files to be linked.
FWIDTH *n*	Total number of columns in the map file.
HEIGHT *n*	Total number of lines on a page in a map file. Refer to the section on map files below.
HWIDTH *n*	Hunk name field width, in columns, in the map file.
INDENT *n*	Number of columns to indent on a line in the map file.
LIB *files*	Specifies the files to be scanned as libraries. Only referenced segments from library files will be included in the final object module. LIBRARY is a valid synonym for LIB.
LIBRARY *files*	Specifies the files to be scanned as libraries. Only referenced segments from library files will be included in the final object module. LIB is a valid synonym for LIBRARY.
MAP *file options*	Specifies a file to which a map is to be written. Options controls which parts of the map will be written. See the MAP section for more information.
MAXHUNK *n*	Limits the maximum size hunk that Blink will create when coalescing hunks. This can be used to control fragmentation of memory. The default size is no limit on hunk size.
ND	Suppresses any HUNK_DEBUG, symbol table information or hunk names in the final object file. This is equivalent to the object file that would be produced if STRIPA were run on the final object file. NODEBUG is a valid synonym for ND.
NOALVS	Prevents BLINK from creating ALVs to resolve 16 bit PC relative code. This can be used to stop BLINK from creating a non-relocatable object from what was intended to be relocatable code.

NODEBUG — Suppresses any HUNK_DEBUG, symbol table information or hunk names in the final object file. This is equivalent to the object file that would be produced if STRIPA were run on the final object file. ND is a valid synonym for NODEBUG.

OVERLAY — Specifies the start of an overlay tree terminated by a line consisting of a single hash sign '#'. See the OVERLAY section for more information.

OVLYMGR *file* — Directs BLINK to use <file> as the Overlay Manager, instead of the default. The file should consist of a single code hunk of name *NTRYHUNK.*

PRELINK — Causes BLINK to output an object module with references and definitions still intact so that it can be linked later on to produce a final executable file. This is designed for development of a large project where the programmer is only changing a single source module. This is described in more detail below. Note that this doesn't function with overlays.

PLAIN — Turns off the FANCY option to map file.

PWIDTH *n* — Width of program unit name fields in the map file.

ROOT *files* — Specifies the object files that are the primary input to the linker. These object files will always be copied to the root of the object module. You must specify at least one object file for the root. If it appears as the first option to BLINK then the ROOT keyword is optional and may be omitted. FROM is an acceptable synonym for ROOT. ROOT may be used more than once with the files for each ROOT adding to the list of files to be linked.

SC — Causes all CODE hunks to be coalesced into a single hunk. SMALLCODE is a valid synonym for SC.

SD — Causes all DATA and BSS sections to be coalesced into a single hunk. This is useful for combining all data hunks from a program into a single hunk, decreasing load time but potentially producing

larger hunks that are difficult to scatter load. SMALLDATA is a valid synonym for SD.

SWIDTH *n* — Width of symbol names field in the map file.

SMALLCODE — Causes all CODE hunks to be coalesced into a single hunk. SC is a valid synonym for SMALLCODE.

SMALLDATA — Causes all DATA and BSS sections to be coalesced into a single hunk. This is useful for combining all data hunks from a program into a single hunk, decreasing load time but potentially producing larger hunks that are difficult to scatter load. SD is a valid synonym for SMALLDATA.

TO *file* — Specifies target object module to create. If omitted it defaults to the same name as the first object module listed on a FROM option with its .o suffix removed.

VER *file* — A destination file to contain all linker output messages. If you do not specify it then all messages go to the terminal. VERIFY is a valid synonym for VER.

VERBOSE — Causes BLINK to print out the names of each file as it processes it.

VERIFY *file* — A destination file to contain all linker output messages. If you do not specify it then all messages go to the terminal. VER is a valid synonym for VERIFY.

WIDTH *n* — Sets the maximum line length for the map and cross reference listings. This is useful when sending the output to a device which has different line length requirements. If not specified it defaults to 80.

WITH *file* — Specifies a file containing BLINK command options to be processed for this link. More than one WITH file may be specified as may WITH files contain WITH statements. The contents of all with files will be treated as if they were specified on the BLINK command line.

XREF *file* — Specifies a file to which the cross reference information will be written. If not specified and a cross reference is requested with the MAP option, the cross reference listing will appear as part of the map file.

Example Command Line Usage

Although BLINK is completely compatible with ALINK *WITH* files the order and format used to input options is quite position independent. FROM files and LIBRARY options can span several lines with or without line continuation characters '+'.

The examples below reflect this remarkable flexibility.

Example WITH files

```
from c.o
vt100.o
init.o window.o
xmodem.o remote.o kermit.o script.o
lib lib:lc.lib lib:amiga.lib
VERBOSE
SMALLCODE
SMALLDATA
to vt100
```

An alternative would look like this:

```
FROM c.o
vt100.o
init.o window.o
xmdem.o remote.o kermit.o script.o
LIBRARY
lib:lc.lib
lib:lc.lib
MAP ram:vt100.map
TO df1:vt100/vt100
SMALLCODE
SMALLDATA
VERBOSE
```

Another alternative would be a type of standard of header WITH file which you can append the specifics to the end of the file and save some typing each time you needed to create a WITH file.

```
LIBRARY
ram:lc.lib
ram:amiga.lib
SMALLDATA
SMALLCODE
MAP
ram:mymap
VERBOSE
XREF ram:myxref
FROM c.o
vt100.0 init.o window.o xmodem.o remote.o kermit.o script.o
```

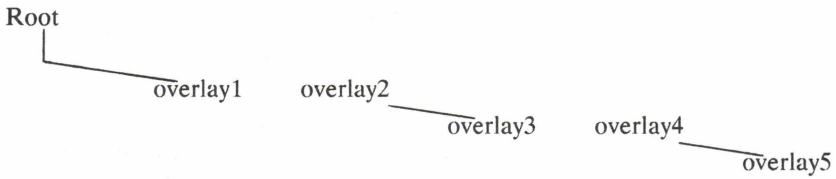
In the above example you will notice that no TO file was specified. BLINK will automatically create a TO file for you in the current directory with the name of the first module in your FROM list not including the startup routine in this case c.o. In the above example the executable will be called VT100 without the .o extension.

Introduction To Overlays

Overlays are code and data hunk groups which reside in main memory only while in use. When one loaded group is no longer needed, and another desired, the memory used by the old group is reclaimed, and may be used to load in the new group. This group loading and unloading is performed automatically.

Overlays on the Amiga are organized as a tree structure, consisting of a permanently resident *root* node, or group of code and data hunks (usually the libraries, your main program, extensively used subroutines, etc.) and one or more *overlay nodes*, or groups of code and data hunks which are to be loaded only as needed, to conserve memory space.

For example:



The above diagram represents an overlay tree. In this tree, overlay nodes 1 and 2 execute independently, using the same memory, one at a time. Also, if overlay node 2 IS loaded, it can then load either of nodes 3 and 4, which share their own block of memory to execute in. Thus, if 3 is loaded and executing, 4 and 1 are NOT loaded, overlay node 2 IS loaded, and overlay 5 MAY be loaded. If overlay node 1 is loaded and executing, then nodes 2 to 5 are NOT currently in memory.

It is NOT necessary that your overlay tree be a binary tree, balanced, or of a depth greater than 1 – you can have 50 overlay nodes, all hanging from the root node. The nodes may be all mutually exclusive and independent of each other, or one long chain of overlay nodes. It is up to the programmer to decide which of routines can operate independently of each other.

Overlays are not a general-purpose mechanism. They are intended primarily for large applications (100 kbytes or more), which must run in a small amount of memory, leaving more for data. Such applications should also have a great deal of modularity, with a definite calling hierarchy among the code routines.

One good example (except for size) is a linker. Although it does not use overlays, it is feasible for BLINK to have overlays. If you run BLINK with the VERBOSE option, you see messages about *Parsing, Pass1*, and so on. These represent clear phases of linking that are independent of each other – the only continuity is in the internal data structures they create and manipulate.

To maximize available memory, either for the linker's data structures or for other concurrent programs and their data, BLINK could have been linked to use overlays, allowing no more than about 5 to 10 kbytes resident in memory at any given moment. The sole overhead would be a slight drop in speed performance, from the individual loading of the separate overlay nodes. Fortunately, BLINK is relatively small and efficient in memory usage, it is not using overlays.

The Overlay Manager

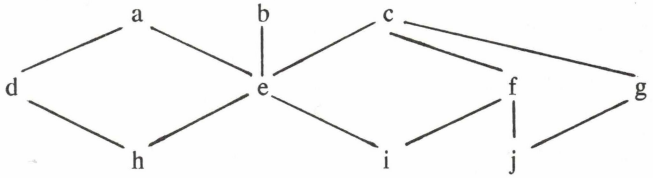
Overlays depend on this routine. Every time a routine in an overlay node is referenced, a call is made to the Overlay Manager, which then scatter loads the desired overlay node, unloading any existing nodes which exist on another, mutually exclusive branch of the tree. You do not need to worry about this, though – these calls to the Overlay Manager are put in for you automatically by BLINK, and are effectively invisible to your own code.

The Overlay Manager can be put into the libraries, rather than kept as a separate file, as under ALINK. You can also override the original Overlay Manager with one of your own, thanks to an extension in BLINK. See the section on Custom Overlay Managers, for more information on this.

Using Overlays

When using overlays, you should design your overlay tree after you have an idea of the calling hierarchy of your application. Those routines which are called frequently, or by almost every other routine in your application, should be placed in the root node. Generally, the more commonly accessed the routine, the closer to the root node it should be. Major independent routines however, should be placed more in accordance with the general calling hierarchy.

Overlay Node Restrictions There are restrictions when referencing routines across overlay nodes. Given the tree:



The following calling restrictions apply:

- Any of the overlay routines (d, e, f, g, h, i, j) can call any of the root node routines (a, b, c).

- Any of the root node routines (a, b, c) can call any of the first overlay level routines (d, e, f, g).

- Routine d can call only root node routines.
- Routine e can call either h or i, in addition to any of the root node routines. But only h or i.
- Routines f and g may call each other, any of the root node routines, or routine j.
- Routines h and i may each call routine e, or any of the root node routines.
- Routine j may call routines a, b, c, f, and g, and only those routines.

Thus if routine a is the main program, and b and c are some type of output routine, and d, e, and f and g were major, mutually exclusive functions, and so on with h, i, and j, you should begin to understand how to go about designing your tree.

If you do not come up with an efficient tree, your program will merely be a little unorganized – an incorrect tree will result in BLINK warning you.

Remember – ALWAYS put your main program into the root node (using either FROM or ROOT). Putting it in an overlay node will not work.

WITH Files

Similar to ALINK, BLINK is uses WITH files to inform the linker that you want overlays – the syntax is the same as for ALINK. Basically, after the keyword OVERLAY, you should list the file or files of your node or nodes. One line (continuation is possible, with an asterisk '*' at the end of the line to continue) per node, as many files as you want per node.

The nodes (i.e. file name groups) should be listed in pre-order fashion (see the section on examples for a clearer understanding of this).

There is one other factor you should be aware of about the syntax – asterisks '*'). Currently, not only are they used individually at the end of a continued WITH file line, but also at the beginning of a new line if the depth of the node being listed is greater than a numeric value of one.

As an example, the tree described an earlier section would look like this in WITH file syntax:

*OVERLAY overlay1 overlay2 *overlay3 **overlay5 *overlay4 #*

Note the '#' – this terminates the list as does an End-Of-File. Also note that since nodes 1 and 2 are at depth 1, they have no asterisks. As a general rule the depth of a node in the tree is represented by depth, less one, prefixed asterisk.

The MAP File

When overlays are used, appropriate sections of the map file will include the information generated by those nodes. Overlay references, and symbols, will appear in the proper map file sections to which they apply.

In specifying the desired map file sections, one specific to overlays is available, and is obtained using the 'O' option to the MAP keyword. This generates a section very much like the hunk section provided by the 'H' option. It has two additional fields which give the overlay level and ordinate of the overlay node to which the code and data hunks belong.

Using the 'O' option and having NO overlays is not a problem since nothing is produced.

Custom Overlay Managers

For advanced programmer the option of writing your own overlay manager exists. This is achieved by a cycle of compile (more likely, assemble) and include it with the command:

OVLYMGR <*file*>

The object file must contain the symbol *_ovlyMgr*. This is EXACTLY how the symbol must appear in the object file – the character case is important. You may need to omit the leading underscore, since Lattice C and the assembler both put it in for you. The code hunk must also have a section name of *NTRYHUNK*.

Where it is necessary to have a code hunk which absolutely MUST be the very first hunk in the file, no matter what the linker tries to do, then you should give it the section name *NTRYHUNK* – BLINK always forces this to be the first hunk. However, this hunk is never merged with any other hunk, and there can only be one such hunk. Also, use of this section name for your own code segment will mean that you can NOT use overlays (unless the segment is your own overlay manager) – try to avoid using with this. You do not have to use *OVLYMGR* to use the section name *NTRYHUNK* – just list it as another object file.

Note that the first hunk has the unenviable privilege of also being THE program entry point.

Little more can be said here about writing your own custom overlay manager. For further information examine hunk 0 of a program which uses overlays. It does exactly what is necessary to do the overlays, and handle the entry point problem.

Compiler Options and Overlays

The -B Option

This option causes all data within a program, whether in a root node or an overlay node (if any), to be merged into one large block which is referenced via 16-bit references. Overlays in BLINK have NO effect on this behavior.

The -R Option

The -R option causes all subroutine references to be via 16-bit relative branches. This modification is extended to overlay nodes, but not across them. Routines in one overlay node being referenced by routines in other overlay nodes are referenced via an *Overlay Call Vector* or *OCV*, which is automatically created for you. An *OCV* is responsible for telling the Overlay Manager to load the desired overlay node.

References within a node which span more than 32k are given an *ALV*, or *Automatic Link Vector*, again, automatically created for you, and guaranteed to transfer control to the actual routine you were trying to reach.

> NOTE: If you use the -R compiler option and overlays, you will find the *OCV*'s being referenced by *ALV*'s when the -R compiler option is used.

Other BLINK Options and Overlays

SMALLCODE

When the SMALLCODE option is used with overlays, BLINK tries to merge as much of the code as possible, without crossing overlay node boundaries. No code will ever move out of the node it was defined to be in.

SMALLDATA When the SMALLDATA option is used with overlays, BLINK merges data hunks together within each overlay node and the root. Data does NOT get moved across node boundaries.

There is an exception – if the -B compiler option was in effect on any or all object files, the data segments of those object files will ALL be moved into the root node.

Overlay Examples

Example 1 – "Hello, World!"

This example is based on the following functions:

```
#include <stdio.h>

void main()
  {
    Hello();
    World();
        }

void SayWord(s)
   char *s;
   {
    printf("%s", s);
                }

void Hello()
   {
    SayWord("\nHello, ");
                    }

void World()
   {
    SayWord("World!\n");
                    }
```

where the first two functions are defined in the file HW.C, and the other two are defined in HELLO.C and WORLD.C, respectively.

While decidedly extravagant for a "Hello, World!" program, it can be used to demonstrate the basic operation of Overlays with BLINK.

After compilation, the object files HW.O, HELLO.O, and WORLD.O are available for linkage. The following WITH file is given to BLINK to build the file:

```
ROOT /lib/c.o hw.o
OVERLAY
hello.o
world.o
#
LIBRARY /lib/lc.lib /lib/amiga.lib
MAP hw.map hxsflo
TO hw
```

which corresponds to the following tree:

```
c.o,hw.o,lc.lib,amiga.lib
      |              |
   hello.o        world.o
```

BLINK evaluates the WITH file, creating an executable called *HW*, and a map file with all possible map information, called *HW.MAP*. The executable consists of a *root* node, containing all of the code and data in *C.O* and *HW.O*, and two separate *overlay* nodes, containing the code and data of *HELLO.O* and *WORLD.O*, respectively. The data and code for the libraries *LC.LIB* and *AMIGA.LIB* are also in the *root* node. To run the program, just do what you normally do to run a program – the overlay operation is automatic, and you will see the message, "Hello, World!" printed out on your display.

Example 2 – A General Editor

Sorry, but there is no source code available for this one. Suppose you had written a new program to edit both graphics and text in a window. Immediately, two overlay nodes should come to mind:

- One node for the graphics-specific editing routines.
- Another for the text-specific routines.

The reason is because if you are busy editing text, you do not need to keep the graphics routines loaded and vice-versa, giving you more memory for your data structures.

Operating the Linker

Another feature of the program would be to print the edited data. For this example, the internal data format is so elegant that you only need one print procedure to print both graphics and text (the text is treated as graphics). Unfortunately, the print routine occupies 60 kbytes of memory. Fortunately, it's only printing – it's only done when requested. Another overlay node is born, probably out of the *root* node. The tree for this application resembles the following:

```
root, with miscellaneous I/O routines and main entry point
        |                    |                |
    graphics               text           printing
```

meaning a WITH file as follows:

```
ROOT /lib/c.o omniedit.o
OVERLAY
graphedit.o
testedit.o
print.o
#
LIBRARY /lib/lc.lib /lib/amiga.lib
TO omniedit
```

Consider a situation where the graphics editor uses various iconic tools to do things with, such as a paint brush, a fine-detail tool, and a flood-fill tool. Only one tool can be used at a time, so why keep the others in memory if spare memory is the prime goal? No good reason, so the routines which support these tools can be overlays hanging off the graphics overlay node, and the graphics node itself can just contain the code for refreshing the display window. This make the tree more like the following:

```
root, with miscellaneous I/O routines and main entry point
      |                    |                |
   graphics              text           printing
for window
refreshing
      |
paintbrush  fine-detail  flood-fill
```

which yields:

```
ROOT /lib/c.o omniedit.o
OVERLAY
graphedit.o
*paintbrush.o
*finedetail.o
*floodfill.o
testedit.o
print.o
#
LIBRARY /lib/lc.lib /lib/amiga.lib
TO omniedit
```

This refinement process could continue for several more levels. The point is that it would easily adapt to overlays, which would minimize the memory requirements of the code of the program, and leave more for the application's data, and for the code and data of other applications running concurrently.

Map Files

The command line structure for these files takes the form:

MAP <map file> <map options>

These files provide a large number of options for the programmer to customize the output format. The *<map options>* are:

F H L O S X FANCY PLAIN

Note the use of just one character on most options. The options may be used individually or combined in the general form:

MAP [[filename],option,option,...]

where:

filename map output file

option letter of report to produce (see below)

The single character options are shown below:

F MapFile

H MapHunk

L MapLibraries

O MapOvly

S MapSym

X MapXref

PLAIN and *FANCY* are used to select the sub-options below:

FANCY flag to allow printer control characters in the map file, *(default condition)*

FWIDTH *n* width of file names *(default 16)*

HEIGHT *n* lines on a page in map file, 0 indicates no pagination *(default 55)*

HWIDTH *n* width of hunk names *(default 8)*

INDENT *n* columns to indent on a line, this is included in width *(default 0)*

PLAIN turns off the FANCY option

PWIDTH *n* width of program unit names *(default 8)*

SWIDTH *n* width of symbol names *(default 8)*

WIDTH *n* columns allowed in map file *(default 80)*

Special Names

The following hunk section names are reserved to have special meanings:

NTRYHUNK There can only be one of these and the hunk with this name will be the FIRST hunk in the output executable. Usually this is used only by the overlay manager's code hunk.

_MERGED Only data and BSS Hunks with this name are permitted. If a data or BSS hunk has this name it WILL be merged with any other hunks with the same name, and moved into the root node (if overlay nodes exist).

_NOMERGE Any hunk with this name will NEVER be merged with any other hunk. It will occupy a single, separate hunk in the final output executable.

The name *BLINKWITH:* is a special ASSIGN name to BLINK – if defined, it is assumed to be a default WITH file. This file appears similar to any other WITH file, and its contents are always applied. However, it does NOT prevent the usage of the command line or another WITH file. Command lines and non-default WITH files do NOT override the default WITH file – they are effectively appended to it.

Thus, creating a default OVERLAY tree is not a good idea and is flagged as an error. The OVERLAY directive should only be used in a non-default WITH file.

Some example default and non-default WITH files look like this:

Example 1.

```
FROM /lib/c.o
LIB /lib/amiga.lib /lib/lc.lib
SMALLCODE  SMALLDATA  BATCH
```

Example 2.

```
FROM maina.o mainb.o
OVERLAY
ov1a.o ov1b.o
*ov2a.o
ov1c.o
#
MAP miffle.map hsflxo
TO miffle
```

Note that the OVLYMGR directive can be used in the default WITH file, although it is not recommended (like OVERLAY).

Terminology

The following terms are used throughout this section:

ALV — Automatic Load Vector. A structure automatically created by BLINK when a 16-bit subroutine reference cannot be expressed in only 16 bits. Instead, the 16-bit reference branches to the *ALV*, which then does a 32-bit to the final destination.

OCV — Overlay Call Vector. A structure automatically created by BLINK when a valid overlay reference is made. A branch to a routine in a child overlay node is patched to branch to an appropriate *OCV*, instead. The *OCV* then invokes the Overlay Manager with the necessary arguments to ensure that the desired routine's node is loaded into memory, and control then passes to the routine.

Overlay Level — The depth within the overlay tree which a given overlay node occupies. The root node is at level 0. All the immediate children are at level 1, and so on.

Overlay Node — A group of code and data hunks, which are loaded via the Overlay Manager, rather than at the application's startup time.

Overlay Ordinate — The ordinal value of the node for the current level. This concept can be difficult to follow. Basically, each parent node has to know what to call each of its progeny. It calls them child 1, 2, 3, and so

on. Another parent overlay node at the same ordinate level, or a different one, uses the same nomenclature of child 1, 2, 3, 4, etc.

Overlay Reference — A subroutine call from one node in an overlay tree to a routine in any of its direct descendant nodes. These type of references go through *OCV's*, and are the only time the overlay manager should ever be invoked.

Overlay Tree — A graphical representation of the hierarchical organization of a group of overlay nodes and their root node, within an application.

Root Node — The group of code and data hunks first loaded when an application is launched. There is always a root node and it is always resident in memory while the program is operating. A non-overlay program is a root node with no overlay nodes.

Section 7. Error Messages

Compiler Error Messages

The Lattice AmigaDOS C Compiler produces three types of error messages:

- Operational Errors
- Syntax Errors and Warnings
- Internal Errors

A detailed discussion of these error messages and their causes follows.

Operational Errors

These indicate that the compiler is having problems operating correctly because it cannot access the required files or there is insufficient disk or memory space.

Syntax Errors and Warnings

These indicate that the compiler is having difficulty comprehending your C source code. The message includes the source file name and line number identifying the point at which the problem was detected. An *'error message'* indicates that the problem prevents the construction of a meaningful object module and must be corrected.

A *'warning message'* indicates that the compiler has detected something unusual but will proceed to make an object module, using appropriate assumptions about your source code intentions.

Internal Errors

These indicate that the compiler encountered some internal condition causing a failure to compile. We would ask you to report it to our Technical Support Department. However, before doing so, we would be grateful if you would conduct a few experiments with your source code to see if you can make the problem disappear.

Operational Errors

The following error messages are displayed when an *'Operational Error'* occurs:

Can't open source file	The first phase of the compiler was unable to open the source file. This error usually occurs because of misspelled file names or failure to specify the correct drive and/or directory path.
Can't create object file	The second phase of the compiler was unable to create the object file. This error usually results from a full directory on the output disk.
Can't create quad file	The first phase of the compiler was unable to create the quad file. This error usually results from a full directory on the output disk.
Can't open quad file	The second phase of the compiler was unable to open the quad file. This error usually occurs when you call the *LC1* command directly with an invalid quad file name.
Combined output file name too large	The output file name constructed by combining the source or quad file name with the text specified using the -o option exceeded the maximum file name size of 64 bytes.
File name missing	A required file name was not specified. For the *LC* and *LC1* commands, you must specify a source file, and *LC2* requires a quad file name.
Intermediate file error	The first phase of the compiler encountered an error when writing to the quad file. This error usually results from an *out-of-space* condition on the output disk.
Invalid command line option	An invalid command line option was specified, and that option will be ignored. Refer to *LC*, *LC1* and *LC2* in Section C for valid command line options.

Invalid symbol definition

The name attached to -d specifying a symbol to be defined was not a valid C identifier or was followed by text which did not begin with an equal sign.

No functions or data defined

The compiler reached the end of the source file without finding any data or function definitions. One very common cause of this error is to forget a comment terminator (*/) during the first function in the source file. This causes the compiler to gobble up your program as if it all were a comment.

Not enough memory

This message is generated when either phase of the compiler uses up all the available working memory. To cure this, you can switch to the "big compiler" via the -B option on the *LC* command. If you are already using the big compiler, then you will need to install more memory or remove some resident program (e.g. Workbench) from your system. An alternative is to break up your source file into smaller pieces. This is easy if the file consists of a lot of small functions. If it consists of a few extremely large functions, you should probably ask yourself if you are making good use of C's excellent support for modular programming.

Object file error

The second phase of the compiler encountered an error when writing to the object file, probably because there is no more space on the output disk.

Parameters beyond file name ignored

Additional information was present on the *LC1* or *LC2* command line beyond the name of the source or quad file to be compiled. A common source of this error is to place compiler options after the file name. This was required on earlier versions of the Lattice C compiler but not on this version.

Unrecognized -c option

One of the characters following the -c option was not a recognized compiler control character. See the

	LC and *LC1* commands for a list of the valid compiler control options.
-i option ignored	More than four -i option strings were specified. Only the first four are retained and used.

Syntax Errors and Warnings

Syntax errors and warnings are reported via a message with the following format:

fff nnn Error xxx: mmm

where the message components are:

fff This is the name of the source file being processed when the error occurred.

nnn This is the source file line number being scanned when the error occurred. Source file lines begin at 1, not 0.

xxx This is the error number, as listed below.

mmm This is the error message, as listed below.

Note that all of these fields are variable length.

All messages indicate *'Fatal Errors'* UNLESS the message number listed below is followed by a (W). When a fatal error occurs, the compiler will not produce a usable object module. The *LC* command alerts you to this condition by beeping and pausing, unless you use the -C option to force continuous compilation.

If the message number below is followed by a (W), then it is a warning. When such a message is displayed, the compiler will produce a usable object module by making reasonable assumptions about what you intended the source file to do. Nevertheless, it is a good idea to investigate these warnings, since the compiler's assumptions may disagree with your intentions.

1 This error is generated by a variety of conditions in connection with pre-processor commands. These include:

- specifying an unrecognized command
- failure to include *white space* between command elements
- use of an illegal pre-processor symbol.

2 The end of an input file was encountered when the compiler expected more data. This may occur on an *#include* file or the original source file. In many instances, correction of a previous error will eliminate this condition.

3 The file name specified on an *#include* command was not found.

4 An unrecognized element was encountered in the input file which could not be classified as any of the valid lexical constructs (such as an identifier or one of the valid expression operators). This may occur if control characters or other illegal characters were detected in the source file.

5 A pre-processor *#define* macro was used with the wrong number of arguments.

6 Expansion of a *#define* macro caused the compiler's line buffer to overflow. This may occur if more than one lengthy macro appeared on a single input line.

7 The maximum extent of *#include* file nesting was exceeded; the compiler supports *#include* nesting to a maximum depth of 4.

8 A *cast* (type conversion) operator was incorrectly specified in an expression.

9 The named identifier was undefined in the context in which it appeared, that is, it had not been previously declared. Note that this message is only generated once; subsequent encounters with the identifier assume that it is of type int (which may cause other errors).

10 An error was detected in the expression following the '[' character (presumably a subscript expression). This may occur if the expression in brackets was null (not present).

11 The length of a string constant exceeded the maximum allowed by the compiler (256 bytes). This will occur if the closing " (double quote) was omitted in specifying the string.

12 The expression preceding the '.' (period) or '->' structure reference operator was not recognizable by the compiler as a structure or pointer to a structure.

13 An identifier indicating the desired aggregate member was not found following the '.' (period) or '->' operator.

14 The indicated identifier was not a member of the structure or union to which the '.' (period) or '->' referred.

15 The identifier preceding the '(' function call operator was not implicitly or explicitly declared as a function.

16 A function argument expression specified following the '(' function call operator was invalid. This may occur if an argument expression was omitted.

17 During expression evaluation, the end of an expression was encountered but more than one operand was still awaiting evaluation. This may occur if an expression contained an incorrectly specified operation.

18 During expression evaluation, the end of an expression was encountered but an operator was still pending evaluation. This may occur if an operand was omitted for a binary operation.

19 The number of opening and closing parentheses in an expression was not equal. This error message may also occur if a macro was badly specified or improperly used.

20 An expression which did not evaluate to a constant was encountered in a context which required a constant result. This may occur if one of the operators not valid for constant expressions was present.

21 An identifier declared as a structure or union was encountered in an expression where aggregates are not permitted. Only the direct assignment and conditional

operators may be used on aggregates, and explicit or implicit testing of aggregates as a whole is not permitted.

22 (W) An identifier declared as a structure or union appeared as a function argument without the preceding '&' operator. In Version 4.0 of Lattice C, aggregates may be passed by value, so this is a legal construction. The warning message is generated to alert you that earlier versions of Lattice C passed the address of the aggregate in this case.

23 The conditional operator was used erroneously. This may occur if the '?' operator was present but the ':' was not found when expected.

24 The context of the expression required an operand to be a pointer. This may occur if the expression following '*' did not evaluate to a pointer.

25 The context of the expression required an operand to be an *lvalue*. This may occur if the expression following '&' was not an *lvalue*, or if the left side of an assignment expression was not an *lvalue*.

26 The context of the expression required an operand to be arithmetic (not a pointer, function, or aggregate).

27 The context of the expression required an operand to be either arithmetic or a pointer. This may occur for the logical OR and logical AND operators.

28 During expression evaluation, the end of an expression was encountered but not enough operands were available for evaluation. This may occur if a binary operation was improperly specified.

29 An operation was specified which was invalid for pointer operands (such as one of the arithmetic operations other than addition).

30 (W) In an assignment statement defining a value for a pointer variable, the expression on the right side of the '=' operator did not evaluate to a pointer of the exact same type as the pointer variable being assigned, i.e., it did not point to the same type of object. The warning also occurs when a pointer of any type is assigned to an arithmetic object. Note that the same message may be a fatal error if generated for an initializer expression.

31 The context of an expression required an operand to be integral, i.e., one of the integer types (*char, int, short, unsigned,* or *long*).

32 The expression specifying the type name for a cast (conversion) operation or a *sizeof* expression was invalid.

33 An attempt was made to attach an initializer expression to a structure, union, or array that was declared *auto*. Such initializations are expressly disallowed by the C language.

34 The expression used to initialize an object was invalid. This may occur for a variety of reasons, including failure to separate elements in an initializer list with commas or specification of an expression which did not evaluate to a constant. Some experimentation may be required in order to determine the exact cause of the error.

35 During processing of an initializer list or a structure or union member declaration list, the compiler expected a closing right brace, but did not find it. This may occur if too many elements were specified in an initializer expression list or if a structure member was incorrectly declared.

36 A statement within the body of a switch statement was not preceded by a case or default prefix which would allow control to reach that statement. This may occur if a break or return statement is followed by any other statement without an intervening case or default prefix.

37 The specified statement label was encountered more than once during processing of the current function.

38 In a body of compound statements, the number of opening left braces '{' and closing right braces '}' was not equal. This may occur if the compiler got *'out of phase'* due to a previous error.

39 One of the C language reserved words appeared in an invalid context (e.g. as a variable name). Note that entry is reserved although it is not implemented in the compiler.

40 A *break* statement was detected that was not within the scope of a *while, do, for,* or *switch* statement. This may occur due to an error in a preceding statement.

41 A *case* prefix was encountered outside the scope of a *switch* statement. This may occur due to an error in a preceding statement.

42 The expression defining a *case* value did not evaluate to an *int* constant.

43 A *case* prefix was encountered which defined a constant value already used in a previous *case* prefix within the same *switch* statement.

44 A *continue* statement was detected which was not within the scope of a *while, do,* or *for* loop. This may occur due to an error in a preceding statement.

45 A default prefix was encountered outside the scope of a *switch* statement. This may occur due to an error in a preceding statement.

46 A default prefix was encountered within the scope of a *switch* statement in which a preceding default prefix had already been encountered.

47 Following the body of a *do* statement, the *while* clause was expected but not found. This may occur due to an error within the body of the *do* statement.

48 The expression defining the looping condition in a *while* or *do* loop was null (not present). Indefinite loops must supply the constant 1, if that is what is intended.

49 An *else* keyword was detected which was not within the scope of a preceding *if* statement. This may occur due to an error in a preceding statement.

50 A statement label following the *goto* keyword was expected but not found.

51 The indicated identifier, which appeared in a *goto* statement as a statement label, was already defined as a variable within the scope of the current function.

52 The expression following the *if* keyword was null (not present).

53 The expression following the *return* keyword could not be legally converted to the type of the value returned by the function.

54 The expression defining the value for a *switch* statement did not define an *int* value or a value that could be legally converted to *int*.

55 (W) The statement defining the body of a *switch* statement did not contain at least one *case* prefix within the statement.

56 The compiler expected but did not find a colon ':'. This error message may be generated if a *case* expression was improperly specified, or if the colon was omitted following a label or prefix to a statement.

57 The compiler expected but did not find a semi-colon ';'. This error generally means that the compiler completed the processing of an expression but did not find a statement terminator. This may occur if too many closing parentheses were included or if an expression was otherwise incorrectly formed. Because the compiler scans through *white-space* to look for the semi-colon, the line number for this error message may be beyond the actual line where a semi-colon was needed.

58 A parenthesis required by the syntax of the current statement was expected but was not found (as in a *while* or *for* loop). This may occur if the enclosed expression was incorrectly specified, causing the compiler to end the expression early.

59 In processing declarations, the compiler encountered a storage class invalid for that declaration context (such as *auto* or *register* for external objects). This may occur if, due to preceding errors, the compiler began processing portions of the body of a function as if they were external definitions.

60 The types of the aggregates involved in an assignment or conditional operation were not identical. This error may also be generated for *enum* objects.

61 The indicated structure or union tag was not previously defined; that is, the members of the aggregate were unknown. Note that a reference to an undefined tag is permitted if the object being declared is a pointer, but not if it is an actual instance of an aggregate. This message may be issued as a warning after the entire source file has been processed if a pointer was declared with a tag which was never defined.

62 A structure or union tag has been detected in the opposite usage from which it was originally declared (i.e. a tag originally applied to a *struct* has appeared on an

aggregate with the union specifier). The Lattice compiler defines only one class of identifiers for both structure and union tags.

63 The indicated identifier has been declared more than once within the same scope. This error may be generated due to a preceding error, but is generally the result of incorrect declarations.

64 A declaration of the members of a structure or union did not contain at least one member name.

65 An attempt was made to define a function body when the compiler was not processing external definitions. This may occur if a preceding error caused the compiler to *'get out of phase'* with respect to declarations in the source file.

66 The expression defining the size of a subscript in an array declaration did not evaluate to a positive *int* constant. This may also occur if a zero length was specified for an inner (i.e. not the leftmost) subscript of an array object.

67 A declaration specified an illegal object as defined by this version of C. Illegal objects include functions which return arrays and arrays of functions.

68 A structure or union declaration included an object declared as a function. This is illegal, although an aggregate may contain a pointer to a function.

69 The structure or union whose declaration was just processed contains an instance of itself, which is illegal. This may be generated if the '*' is forgotten on a structure pointer declaration or, due to some entangled structure definitions, the structure actually contains an instance of itself.

70 The formal parameter of a function was declared illegally as a function.

71 A variable was declared before the opening brace of a function, but it did not appear in the list of formal names enclosed in parentheses following the function name.

72 An external item has been declared with attributes which conflict with a previous declaration. This may occur if a function was used earlier, as an implicit *int* function,

and was then declared as returning another type of value. Functions which return a type other than *int* must be declared before they are used so that the compiler is aware of the type of the function value.

73 In processing the declaration of objects, the compiler expected to find another line of declarations but did not find one. This error may be generated if a preceding error caused the compiler to *'get out of phase'* with respect to declarations.

74 (W) A string constant used as an initializer for a *char* array defined more characters than the specified array length. Only as many characters as required needed to define the entire array are taken from the first characters of the string constant.

75 An attempt was made to apply the *sizeof* operator to a bit field, which is illegal.

76 The compiler expected, but did not find, an opening left brace in the current context. This may occur if the opening brace was omitted on a list of initializer expressions for an aggregate.

77 In processing a declaration, the compiler expected to find an identifier which was to be declared. This may occur if the prefixes to an identifier in a declaration (parentheses and asterisks) are improperly specified, or if a sequence of declarations is listed incorrectly.

78 The indicated statement label was referred to in the most recent function in a *goto* statement, but no definition of the label was found in this function.

79 (W) More than one identifier within the list for an *enumeration* type had the same value. While this is not technically an error, it is usually of questionable value.

80 The number of bits specified for a bit field was invalid. Note that the compiler does not accept bit fields which are exactly the length of an Amiga *word*. These must be declared as an ordinary *int* or unsigned variables.

81 The current line contains a reference to a pre-processor symbol that is a circular definition.

82 The size of an object exceeded the maximum legal size for objects in its storage class. Alternatively, the last object declared caused the total size of declared objects for that storage class to exceed the maximum.

83 (W) An indirect pointer reference (usually a subscripted expression) used an address beyond the size of the object used as a base for the address calculation. This generally occurs when an expression makes reference to an element beyond the end of an array.

84 (W) A *#define* statement was encountered for an already defined symbol. The first definition is *'pushed'*, so that an additional *#undef* statement is needed to undefine the symbol.

85 (W) The expression specifying the value to be returned by a function was not of the same type as the function itself. The value specified is automatically converted to the appropriate type; the warning merely serves as notification of the conversion. The warning can be eliminated by using a *cast* operator to force the return value to the function type. This warning is also issued when a *return* statement with a null expression (i.e. no return value) appears in a function which was not declared *void*; generation of the warning for this particular context can be disabled using the -c option on the *LC* or *LC1* command.

86 (W) The types of the formal parameters declared in the actual definition of a function did not agree with those of a preceding declaration of this function with argument type specifiers.

87 (W) The number of function arguments supplied to a function did not agree with the number of arguments in its declaration using argument type specifiers.

88 (W) The type of a function argument expression did not agree with its corresponding type declared in the list of argument type specifiers for this function. Note that the compiler does not automatically convert the expression to the specified type; it issues this warning instead.

89 (W) The type of a constant expression used as a function argument did not agree with its corresponding type declared in the list of argument type specifiers for that function. In this case, the compiler does convert the expression to the expected type.

90 The type specifier for an argument type in a function declaration was incorrectly formed. Argument type specifiers are formed according to the rules for type names in *cast* operators or *sizeof* expressions.

91 One of the operands in an expression was of type *void*; this is expressly disallowed, since *void* represents no value.

92 (W) An expression statement did not cause either an assignment or a function call to take place. Such a statement serves no useful purpose, and can be eliminated; usually, this error is generated for incorrectly specified expressions in which an assignment operator was omitted or mistyped.

93 (W) An object with local scope was declared but never referenced within this scope. This warning is provided as a convenience to warn of declarations that may no longer be needed (for example, if the code in which the variable was used is eliminated but not its declaration). It may also occur if the only use of the object is confined to statements which are not compiled because of conditional compilation directives such as *#ifdef* or *#if*.

94 (W) An auto variable was used in an expression without having been previously initialized by an assignment statement or appearing in a function argument list with a preceding '&' (i.e. its address passed to a function). Note that the compiler considers the variable initialized once any statement causes it to be initialized, even though control may not flow from this statement to other subsequent uses of the variable. Note also that this warning will be issued if the third expression in a *for* statement uses a variable which has not yet been initialized, which may be incorrect if this variable is initialized inside the body of the *for* statement.

Internal Errors

These errors are reported via the message:

CXERR: xx

where *xx* is the error number.

When such a message occurs, compilation is terminated immediately, and both the quad file and the object file are probably unusable.

1 Invalid error or warning message code number.

2 The compiler has internally called a function which is not applicable to the current host-target environment.

3 Invalid symbol table access.

4 Declaration chain is broken.

5 An unlink error occurred while processing an *undef* command.

6 The compiler attempted to push back too many tokens.

7 There is no aggregate list for a structure reference.

8 Stack underflow has occurred.

9 Invalid attempt to generate the address of a constant.

10 A test value is not a constant.

11 Invalid unary operator.

12 Invalid binary operator.

13 A scaling object is not a pointer or array.

14 An unexpected end-of-chain occurred while restoring internal context.

15 Invalid quad type.

16 Deletion length is less than two bytes.

17 Insufficient memory.

18 An error occurred when releasing memory.

19 Invalid condition during temporary assignment.

20 Invalid condition while processing program section.

21 Literal pool generation error.

22 Invalid condition while processing data section.

23 Invalid quad file.

24 End-of-file while processing '*for*' quad.

25 Invalid register number.

26 Temporary save or restore error.

27 Invalid operand size.

28 Invalid storage base.

29 Error during branch folding.

30 Error during control statement processing.

31 Error during special addressing setup.

32 Invalid object description block offset.

33 Too many function parameters.

34 Indirect argument for call-by-reference.

35 Invalid external relocation value.

95 Function is not declared.

96 No function prototype for function called.

97 Unrecognized #pragma operand.

98 No name for #pragma, libcall or syscall.

99 Bad library base for #pragma.

100 Invalid data for #pragma, libcall or syscall.

101 Short value converted to a pointer.

BLINK Error Messages

1 Parser failure.

2 Pass 1 failure.

4 Coalesce failure.

5 Map failure.

3 Pass 2 failure.

6 Symbol Initialization failure.

200 Out of memory!!

300 Read error on file '%s'

301 Write error on file '%s'

400 *** Break: Blink terminating.

425 Cannot find library %s

426 Cannot find object %s

443 '%s' is an invalid file name.

444 Hunk_Symbol has bad %d symbol %s

445 Invalid HUNK_SYMBOL %s

446 Invalid symbol type %d for %s

447 %s is a load file.

448 Invalid hunk #%d

449 No hunk_end seen for %s

501 Invalid Reloc 8 or 16 reference.

502 Distance for Reloc16 > 32768

503 Distance for Reloc8 > 128

504 Distance for Data Reloc16 > 32768

505 Distance for Data Reloc8 > 128

506 Cannot locate resolved symbol %s

507 Unknown Symbol type %d

508 Symbol type %d unimplemented.

509 Unknown hunk type %d in Pass 2.

510 % symbol - Reference to unmerged data item.

511 Cannot reference '%s' outside defining overlay.

512 Invalid branch to '%s' in overlay node '%s'

513 Multiple 'NTRYHUNK' segments not permitted.

514 Overlay manager '_ovlyMgr' is undefined.

600 Invalid command '%s'

601 %s option specified more than once.

602 Unable to open map file '%s'

603 %s is not a valid number.

604 WITH file is not readable.

605 Cannot open with file '%s'

606 Unable to open output file '%s'

607 No FROM/ROOT files specified.

608 Premature EOF encountered.

609 Error seeking in file %s

610 '%s' has no parent in overlay tree.

999 Unknown internal error, please inform Lattice technical support department.

Section 8. Version Differences

Introduction

This appendix lists the differences between the various Versions 3.xx and Version 4.0 of the Lattice AmigaDOS C Compiler. As always, we've tried to maintain compatibility at both the source level and the object code level. Nevertheless, we could not avoid the following incompatibilities:

ALINK	The Lattice run-time support libraries as distributed by Lattice are not compatible with the Amiga linker *ALINK* from *Metacomco Ltd.*
Floating Point	You must re-compile any program using floating point because the run-time support library was changed. If you link an old object module with the new libraries, pointers may not be preserved across floating point operations.
Function Arguments	You must re-compile any programs that pass structures as function arguments, and you must also re-compile any programs that expect structures as function arguments.
Structure Pointers	You do not need to re-compile programs that pass or expect structure pointers as arguments.
-b Option	If you re-compile a module with the -b option, you must either use the -y option, or re-compile all programs that directly or indirectly call functions in that module. Programs compiled with the -b option expect the data section base address to be in A4. Programs compiled previous to this release will not properly propagate this value.
Using *feof, ferror or fileno*	You must re-compile any program using the *feof, ferror,* or *fileno* macros from *stdio.h.*

Internal Tables

Several internal tables and functions have changed. Most users should not be accessing these directly, since they are not described in the normal documentation and we have never guaranteed to preserve them from one release to another. However, if you have obtained the Version 3.03 or 3.10 library source code and have determined how to use these items, you should now obtain the Version 4.0 library source and check that the items you are using have not changed. In particular, note that the *_pfmt* and *_sfmt* functions have changed, as have the *UFB* and *_iobuf* structures.

I/O Data Structures

If your code is descending into the Level 2 and Level 1 I/O data structures to obtain such things as AmigaDOS file handles, you should note that this area of the library has changed considerably. In particular, the *open* function now returns an AmigaDOS file handle, so you should not need to examine the *UFB* structure in order to obtain that information.

Level 1 I/O Structures

Because of the changes in the Level 1 I/O, it is no longer possible to use file descriptors 0, 1, or 2 to refer to *stdin, stdout,* or *stderr.*

Break Characters

Handling of the break characters CTRL-C and CTRL-D has changed since the last release. Previously, the function *Chk_Abort* could be called to determine if a break character had been pressed. Processing of these characters is now controlled by the *onbreak* function, and the UNIX *signal* function. The function *chkabort* may be called to force checking for these characters, but it does not return any values. Normally checking is performed during I/O calls.

Summary of Compiler Changes

1. The standard header files have been modified to include argument types in their external declarations. A new header file, *STDLIB.H* contains declarations for functions not covered in the other standard headers.

2. The compiler now consults the AmigaDOS logical name *INCLUDE:* to locate files mentioned on *#include* lines.

If the file name is enclosed in quotes "...", then the search starts in the current directory, proceeds through the directories mentioned on the -i options, and finishes in the directory associated with the *INCLUDE* logical name. If the file name is enclosed in angle brackets <...>, then the search is restricted to the *INCLUDE* directory.

3. An additional compatibility code, *'t'* has been added to the -c option on *LC* or *LC1*. This code disables warning messages for structure and union tags that are used without being defined. For example:

   *struct XYZ *p;*

   would normally produce a warning message if the structure tag *XYZ* were not defined.

4. Several of the compile-time limits have been increased, as follows:

LIMIT	OLD	NEW
Input line size	256 bytes	512 bytes
Substitution text size	256 bytes	512 bytes
Macro arguments	8 args	16 args

   Because of these size increases, some very large source programs may cause the compiler to abort with an *'Out Of Memory'* message.

5. The Amiga has two areas of memory designated *CHIP* memory and *FAST* memory. *CHIP* memory is that which is accessible by the 68000 *and* the custom hardware such as the Blitter and Copper for DMA operations. *FAST* memory is the area of the processor address space which is *only* directly accessible by the 68000 and it is added to the system by connection to the Expansion Port by the user.

   The reasoning behind this segmentation of memory is to avoid an overhead in address bus contention between the custom hardware and the 68000 processor for accesses to *FAST* memory. This means that data designated for use with the custom hardware, i.e. screen or audio data, must be located in *CHIP* memory in order that the custom hardware can locate and utilize it. Nevertheless, this also means that the program code need not be located in *CHIP* memory, indeed it would be wasteful of

the system's resources to do so. This has the advantage that access by the 68000 to the program code is not constrained by competition between the processor and the custom hardware since the former has unrestricted access to *FAST* memory. By default the system will load *everything* from the disk into *FAST* memory, when this is available.

The Amiga Object File Format includes a means of specifying which type of memory each program segment is to be loaded into. The Lattice AmigaDOS C Compiler supports this option. There are two command line flags used with phase 2 of the compiler to specify which segments are to be loaded into specific memory.

FLAG	DESCRIPTION
-cc	*chip* memory code
-cd	*chip* memory data
-cb	*chip* memory bss
-hc	*fast* memory code
-hd	*fast* memory data
-hb	*fast* memory bss

The -c flag specifies that the designated sections are to be loaded ONLY into *CHIP* memory. Because of a conflict with the compatibility flag on phase 1, when using the *LC* driver program this option is referenced via the -a flag.

The -h flag specifies that the sections designated are to be loaded ONLY into *FAST* or high-speed memory. Code and data fetches from this memory generally are faster because of the lack of bus contention with screen or audio DMA operations. Any or all of the segments may be specified with each flag.

For example, the following causes all the program *code* to be loaded into *CHIP* memory and all the program *data* and *bss* segments to be loaded into *FAST* memory:

-cc -hdb

Consider this example:

-cd -hc

Experienced Amiga programmers will realize that the first example given above would be completely nonsensical on a machine with expanded memory. The second example is more realistic, particularly if the application is primarily for audio or graphics.

On an unexpanded machine, the complete application (i.e. code, data and bss) has to be loaded into *CHIP* memory, thus the question of these flags does not arise. Software developers are strongly advised to make available two versions of their application – one compiled for use with an unexpanded system, the other for expanded memory systems.

The compiler does not check to determine if you have previously specified the segments to be loaded into different ram. It will simply take the specification that occurs last.

6. The Amiga Load Module Format allows for each program segment to be loaded into separate memory locations. This effect is known as *scatter-loading*, and is very efficient at utilizing fragmented memory. However, this can cause program loading to be excessively time consuming, particularly from a floppy disk. The Amiga linker has the capability to merge similarly named segments. While this may reduce the efficiency of the memory utilization, it can sometimes significantly improve load times.

   The -s command line option for Phase 2 of the compiler is used to control the names of the segments. Used by itself, it causes segments to be named with the default names of code, data and udata. It may also be followed with a section name specification, allowing each segment to be individually named. The formats for these specifications are:

   - c = codename
   - d = dataname
   - b = bssname

   where *codename* is the name to be used with the code section, *dataname* is the name to be used for the initialized data section and *bssname* is the name to be used for the uninitialized data section.

An example command line might contain the following options to name all three segments:

```
-sc=codename -sd=dataname -sb=bssname
```

7. There is a single debug flag -d which is used with phase 1 of the compiler. It causes symbol hunks to be included in the object file for debugger support and debug hunks containing line number information to be included for *OMD* support.

   Because of the amount of extra information in the files, use of the -d option can cause object and load module files to be significantly larger than they would be without it. Once programs have been debugged they must be recompiled without the -d option to reduce their size. This not only releases disk space but also signifies a more efficient approach to programming.

8. The compiler supports the Motorola Fast Floating Point (FFP) format directly. This means that floating point operations compiled with this option will automatically use the Amiga supplied routines to carry out the operations in fast floating point format. No source conversion is necessary. This option is used in the command line for Phase 1 of the compiler, and uses -f as the flag.

   FFP is a unique implementation of floating point specifically for the 68xxx series and is incompatible with the IEEE floating point format. It is a 32 bit floating point representation that does not support double precision. Use of the -f flag with Phase 1 causes the compiler to generate 32 bit FFP storage types and constants.

   It is not possible to mix FFP and IEEE format operations. The support implemented by the Lattice compiler requires the program to be linked with ONLY the appropriate math library. If the -f flag is used, the library *LCMFFP.LIB* must be included before *LC.LIB* in the linker command line. If the flag is ommitted the library *LCM.LIB* must be included before *LC.LIB*.

   Library opening and closing is performed automatically when needed. The libraries are not opened until an operation which requires the library is performed. This may cause a slight pause during the first execution of a floating point operation (and the first FFP format transcendental function) which will affect benchmark timings.

   Note that there is a limited set of transcendental functions available for the FFP

format. The library only supports those routines supplied to us by Commodore-Amiga at the time of this release.

9. The -b flag has been added to *LC* and *LC1*. This flag directs the compiler to generate data references as offsets from a base address register. This is referred to as *base relative addressing*. This change involved extending the Amiga Object File Format with the addition of new relocation types, and forcing the names of the data and bss hunks to be _MERGED.

   This special name is recognized by BLINK and causes these hunks to be coalesced into a special data section addressed by register A4. This option overrides the use of -s. Because of the non-standard relocation types involved, modules compiled with the -b flag are NOT compatible with the original Amiga linker from Metacomco named ALINK.

   There are potential problems when using this option with programs that will be using the *AddTask()* function or may be used as an interrupt handler. Programs compiled with the -b option will be expecting the data section base address to be in address register A4. However, interrupt handlers and tasks created with *Addtask()* will potentially have random values in the registers. With these types of programs you MUST use the -y option for phase 2 which causes register A4 to be loaded with the base address. See the *LC* command in Section C for a complete description.

   It is possible to mix modules using base relative addressing with modules that do not. Those routines compiled with the -b flag will have their data merged into the special A4-addressable section, while those that were not will continue to be scatter-loaded.

   This option is not compatible with the -c option for phase 2. If this option is used for either data or bss, it will override the -b option.

10. Programs that pass structures as function arguments now copy the entire structure onto the stack before calling the function. Previously a pointer was passed to the called function, which then used the pointer to produce a local copy of the structure.

Summary of Library Changes

Translated Mode

The translated mode for Level 2 I/O (*fopen, fclose,* etc.) has been improved so that file positioning works exactly as in non-translated mode. Previously it was not possible in translated mode to obtain a file position via *ftell* and then reposition to the same spot via *fseek*.

This improvement required a slight change in *stdio.h*. The change is invisible to programs which do not use the *feof, ferror,* and *fileno* macros and do not directly reference the *_iobuf* structure.

open Function

The *open* function now returns an AmigaDOS file handle instead of the *UFB* array index.

Level 0 Functions

Earlier versions of the library included a set of I/O functions which interfaced directly with AmigaDOS using file handles. These *'level 0 I/O functions'* were given peculiar names and were only used internally within the library. We have now made the level 0 functions available under the names *dcreat, dcreatx, dopen, dclose, dread, dwrite,* and *dseek*. These functions can be used to bypass the UNIX-compatible I/O services which we provide at levels 1 and 2.

However, if you are already accessing a file via Level 1 or Level 2 I/O, do not use these Level 0 functions because the higher level services will become confused. This warning applies only to the functions mentioned immediately above. The other file handle functions mentioned below can generally be used in conjunction with Level 1 and Level 2 I/O.

New I/O Functions

Some new I/O functions have been added to provide easy access to AmigaDOS file features. The *getfa* function allows you to get some AmigaDOS file attributes. Similarly, *getft* gets the date/time value stored in the file's directory entry. The *getcd* and *getcwd* functions get the current directory path.

See the Index of External Names for additional AmigaDOS functions.

UNIX, XENIX Simulations

Several new functions have been added to provide a more exact simulation of the UNIX and XENIX environments. These include:

abort	*access*	*asctime*
assert	*chdir*	*chmod*
ctime	*fdopen*	*getcwd*
gmtime	*localtime*	*memccpy*
memchr	*memcmp*	*memcpy*
memset	*mkdir*	*perror*
qsort	*rmdir*	*tzset*

See the Index of External Names for additional UNIX/XENIX functions.

printf Function

The *printf* function has been greatly enhanced, with the addition of several format specifications and the elimination of the 200 character output string length limitation.

argv Pointers

Under AmigaDOS, *argv* will either contain a pointer to an array of argument string pointers, or a pointer to the Workbench startup message. See the function description for *main* in the External Name Reference section for a complete description.

Function Consistency

Several library functions have been renamed in order to make them consistent with the rest of the library. These are:

OLD NAME	NEW NAME	DESCRIPTION
mkext	*strmfe*	Make file name with extension
mkname	*stcgfn*	Get node part of file name

Control-C Handling

Unlike the previous version, control-c handling is automatically enabled in programs produced with this release. To disable control-c, you must modify *_main.c* or provide your own handler via the functions *onbreak* or *signal*.

Stack Overflows

In general, the library routines are generated WITHOUT stack overflow checking. However, because of the enormous stack overhead required by AmigaDOS, there is a separate I/O interface object module provided which will check to determine if the required 1K of stack space is available before calling AmigaDOS. This is not part of the standard library because stack checking does not work correctly if a portion of the program is set up as a separate task using the ROM Kernel *AddTask()* function. If the program will not be performing this type of multi-tasking, however, and file I/O will be required, it is highly recommended that this file *IOS0.O* be included with the other object modules when linking.

Console Handling

Because a task running on the Amiga may not necessarily have an AmigaDOS console window attached to it, the default handlers for stack overflow and control-c do not attempt to output a message to the console. These routines now use an Intuition requester to notify the user of the exception and prompt for the appropriate response. The control-c handler allows the user to continue execution if desired. The stack overflow handler resets the stack to a known state, then waits for an acknowledgment before exiting.

Both handlers attempt to exit cleanly, closing any open files and de-allocating any memory allocated during program execution. This is only effective on files opened with Level 1 or Level 2 I/O function calls, and memory allocated with the Lattice run-time library memory allocation routines. The library now includes the function *onexit* that can be used to establish a function which will be called when the program terminates. If the program will be opening files or allocating memory via direct system calls then a cleanup function should be established using *onexit* to correctly handle these system resources.

Math Libraries

Because of the addition of Fast Floating Point support, the math functions have been separated from the rest of the run-time library. This means there are now three Lattice libraries:

LC.LIB	I/O, memory allocation, string functions, etc.
LCM.LIB	IEEE math functions
LCMFFP.LIB	Motorola FFP math functions.

Linking Math Libraries

If no floating point operations are involved in the program, then *LC.LIB* and *AMIGA.LIB* are the only libraries required. However, if the program uses floating point operations (even just *printf* of a floating point value), then one of the math libraries MUST be included in the linker's library specification BEFORE the other two libraries. Which library to use depends on whether the ffp option -f was used. It is NOT possible to use both libraries. You cannot mix IEEE and FFP formats in a program.

Note that this means all object files in a load module, not just all functions in a source file. To link a program which uses no floating point:

blink FROM c.o,file.o TO file LIB lc.lib,amiga.lib

To link a program which uses IEEE floating point:

blink FROM c.o,file.o TO file LIB lcm.lib,lc.lib,amiga.lib

To link a program which uses FFP floating point:

blink FROM c.o,file.o TO file LIB lcmffp.lib,lc.lib,amiga.lib

IEEE Math Routines

The IEEE floating point and integer math routines have been optimized, providing an improvement in speed by a factor of 5 to 10.

Library Base Vectors

The Amiga library base vectors:

- DOSBase
- IntuitionBase
- MathBase
- MathTransBase

- DiskfontBase
- GfxBase

are now part of the library. They may be safely referenced as externals in order to avoid possible duplicate declarations by unrelated functions.

Miscellaneous Changes

This release of the AmigaDOS C compiler as distributed by Lattice includes the following utilities:

ASM	An Amiga compatible 68000 macro assembler.
BLINK	An Amiga compatible linker.
LCOMPACT	A utility to compress header files.
MEMWATCH	A utility which observes the lower part of memory for errors.
OML	An object module librarian.
POPCLI	A windows utility

Also, since the libraries distributed by Lattice are not compatible with ALINK and the compiler is capable of producing files incompatible with ALINK, this linker is no longer distributed by Lattice as part of the compiler package.

Lattice supported the development of BLINK as a replacement for ALINK and will continue to develop and enhance this linker.

It is now possible to invoke the compiler, librarian, and linker with a single command. See the description of the *LC* command in Section C for a complete description.

Section 9. Compiler Functions

Introduction

Compiler commands and external names are contained within the following Appendices:

Appendix C: Commands	Describes the AmigaDOS commands which are provided with the compiler package.
Appendix D: Data Names	Describes the public data names which are defined in the Lattice libraries.
Appendix E: Environment Variables	Describes the AmigaDOS environment variables used by the compiler and related programs.
Appendix F: Function Names	Describes the public function names which are defined in the Lattice libraries.
Appendix G: Combined Reference	A combined alphabetical listing presenting the name, purpose, page and type of all command and external names covered in the previous appendices.

Some general information on each of these sections is presented below.

Commands

Appendix C describes the AmigaDOS commands which are used to invoke the compiler and the various utility programs.

When you install the compiler, the driver program *LC* is usually copied into the C: command directory. Most operations, including compiling, library updating and linking can be performed via the *LC* command. Most people would then add the appropriate assignment statements to their startup-sequence file in the S: directory so that *LC* can easily locate all the programs and files it requires. See Appendix E for descriptions of the logical name assignments used by *LC*.

Commands are invoked in the following way:

name <input >output options arguments

where all fields except name are optional. The fields are defined as follows:

name — This is the command name. AmigaDOS makes no distinction between upper and lower case in this field.

<input — This field, if present, specifies the file which is to be assigned as this command's standard input. Within a Lattice C program, the standard input file is specified by the Level 2 I/O file pointer named stdin. The input field can be any valid AmigaDOS file or device reference, and no distinction is made between upper and lower case.

>output — This field, if present, specifies the file which is to be assigned as this command's standard output. Within a Lattice C program, the standard output file is specified by the Level 2 I/O file pointer named stdout.

options — This optional field contains one or more *'option specifiers'* with blank separators. In its simplest form, an option specifier consists of a dash (-) followed by a single letter. For example, the -v option on the *LC* command tells the compiler not to generate stack-checking code. Some options contain variable information, which should immediately follow the option letter. For example, -oDH0: on the *LC* command tells the compiler to place the object file on the DH0: disk drive.

arguments — This field, if present, consists of one or more strings which will be passed to your main program via the argv array, as described in Appendix F. The arguments are separated by white space. Any argument containing white space must be enclosed in double quotes, and you can use "" to include a double quote within such an argument.

> NOTE: Some commands are sensitive to the case of the option letter. Also note that our commands use options in the manner of UNIX commands, with leading dashes.

Data And Function Names

Appendices D and F describe the external names (i.e. data and functions) which you will probably need to access from your C programs.

In general, you should not define data and function names which are identical to these external names, because your names will override those in the libraries. This is particularly important when the library calls upon itself, your override may interfere with correct library operation.

For example, if you replace the *strcmp* function with one that behaves differently, the *tqsort* function will not operate, because it calls *strcmp*. Note that if you do duplicate one of the Lattice external names, the linker will not usually report this as an error.

Portability Classifications

Many C programmers are have some concern about portability, in the sense of they do not want to write software which will lock them into a particular hardware system. In addition, professional programmers prefer to design general-purpose modules (i.e. *'software tools'*) which they can re-use from one application to another. At the same time, C is popular because it allows the programmer to write programs approaching the performance of assembly language but with considerably less effort and not least, with fewer maintenance problems.

The Lattice AmigaDOS library attempts to strike a reasonable balance between portability and performance. It includes the highly portable functions defined by the ANSI C Standards Committee, and if you use only these functions, your code will be relatively easy to implement on a broad spectrum of machines.

The library also includes a large number of functions which utilize AmigaDOS more efficiently than the ANSI functions would allow or unique functions which provide access to features which are peculiar to AmigaDOS.

It is your decision to select which of these facilities are the best choice for your own application. To assist you in this decision, each data or function name has been given a portability classification, as follows:

AMIGA	The name is defined by Lattice, but it is only available under AmigaDOS.
ANSI	The name is defined in the proposed ANSI Standard for the C language, and our implementation conforms to this definition.
LATTICE	The name is defined in Lattice's portable library. That is, we guarantee that it will be available on any system for which we provide a C compiler package.
UNIX	The name is defined in AT&T's UNIX System V, and our implementation conforms to this definition. We use UNIX on our VAX computers and on our other minis to verify compatibility.
XENIX	The name is defined in Microsoft's XENIX, and our implementation conforms to that definition. We use XENIX on an IBM-AT computer to verify compatibility.

Function Calling Sequences

It is very important that you understand how we are presenting function calling sequences. In the synopsis section of each function description, we show the general form of the function call, such as:

```
fh = open(name,mode,prot);
```

Then we indicate what kind of information the function expects to receive as its arguments and pass back as its return value. For the open function, this is expressed by:

```
int fh;          file handle
char *name;       file name
int mode;        access mode
int prot;        protection mode
```

The caller must supply arguments which conform to the expectations of the function. Novice C programmers sometimes misinterpret our presentation and conclude that they must declare their arguments exactly as we show in the synopsis. This can lead to horrors such as:

```
int fh,mode,file,prot;
char *name;
name = "myfile";
mode = O_RDONLY;
prot = 0;
fh = open(name,mode,prot);
file = fh;
```

This is both wasteful of memory and totally unnecessary. A much more straightforward approach would be:

```
int file;
file = open("myfile",O_RDONLY,0);
```

In other words, the arguments can be expressed any form you like, as long as they arrive at the function in the correct form. Of course the function does not know, or indeed care, about the name you give its return value.

However, in order for the return value to be handled correctly, you must make certain that the function is declared correctly. For example, this sequence of code will produce bizarre results, assuming there is no declaration for the sin function:

```
double value;

value = sin(1.7);
```

Why? Because in the absence of a declaration, the compiler assumes that sin returns an integer, and so code is generated to convert an integer result into a double for assignment to value. Since sin actually returns a double, the conversion is unnecessary and produces garbage. The correct code is:

```
double value,sin();

value = sin(1.7);
```

This alerts the compiler that sin returns a double, which does not then need to be converted for the assignment.

An even better way to ensure that the standard Lattice functions get declared correctly is to include the appropriate header files into your program. Whenever a special header file

should be used in conjunction with a particular function or variable, we show the appropriate #include statement in the synopsis. The header file will contain any definitions (e.g. constants and structures) which are normally required when you interface with the function or variable. Also, it will contain the necessary external declarations, as discussed in the next section.

Argument Type Checking

The compiler can check which functions are called with the appropriate type of arguments. This feature is automatically activated when you declare an external function with an argument type list, such as:

```
extern int read(int, char *, unsigned);
```

which indicates that the read function should be called with three arguments:

- an integer
- a character pointer
- an unsigned integer

The various header files contain such declarations for the standard Lattice library functions. However, these declarations can be bypassed if you define the preprocessor variable NARGS before including any header files. This can be done on the *LC* or *LC1* command line as:

```
lc -dNARGS myprog
```

or you can include the line:

```
#define NARGS
```

at the beginning of your source file. Note that *NARGS* must be capitalized.

File Names

For all functions which accept file name arguments, you can use the *'complete path'* form unless explicitly noted otherwise. The complete path form is:

drive:path/node.ext

where *drive* is a drive or disk volume name, *path* is a directory path, *node* is a node name, and *ext* is an extension.

The directory path can be a list of node.ext constructs separated by slashes. Lattice functions that generate or parse file names use the character defined in SLASH as the separator. This character defaults to a slash.

Here are several examples of valid file names:

myprog
myfile.c
df1:myprog/pdq/myprog
df0:/pdq/xyz.dir/myfile.c

Environment Variables

Appendix E describes the AmigaDOS logical name assignments which are used by the Lattice compiler and related programs.

Initially, all logical name assignments are constructed by AmigaDOS when the system is first loaded. For example:

- C: is the logical name assigned to the command directory.
- LIBS: is the logical name assigned to the directory containing run-time libraries.

Additional assignments can be made with the assign command. Logical names can be assigned to any disk volume, directory or file.

The most popular use for logical names is to specify the directories where certain file types can be found. They can also be used as shorthand references to the files themselves. For instance, assigning CC: to the file DF1:lc allows you to refer to the file *LC* on the volume in drive DF1: as CC:.

Appendix A. Bibliography

C Programming References

Allan, Boris [1986]

Introducing C, ISBN 0-00383-105-1 William Collins Sons & Co.Ltd., 8 Grafton Street, London W1X 3LA, England.

Bolsky, M.I. [1985]

The C Programmer's Handbook, ISBN 0-13-110073-4, Prentice-Hall, Inc., Englewood Cliffs, NJ 07632, USA.

Brand, Kim Jon [1985]

Common C Functions, ISBN 0-88022-069-4, QUE Corporation, Indianapolis, IN 46250, USA.

Gehani, Narain [1985]

Advanced C: Food For The Educated Palate, ISBN 0-88175-078-6, Computer Science Press, Rockville, MD 20850, USA.

Hunt, William James [1985]

The C Toolbox, ISBN 0-201-11111-X, Addison-Wesley Publishing Company, Reading, MA, USA.

Kernighan, Brian W. and Dennis M. Ritchie [1978]

The C Programming Language, ISBN 0-13-110163-3, Prentice-Hall, Inc., Englewood Cliffs, NJ 07632, USA.

Purdum, Jack [1986]

C Programming Guide - 2nd Edition, ISBN 0-88022-157-7 QUE Corporation, Indianapolis, IN 46250, USA.

Lawrence, David and Mark England [1985]

The C Compendium, ISBN 0-946408-86-6, Sunshine Books, 12-13 Little Newport Street, London WC2H 7PP, England.

Purdum, Jack [1986]

C Programming Guide - 2nd Edition, ISBN 0-88022-157-7 QUE Corporation, Indianapolis, IN 46250, USA.

Purdum, Jack [1985]

C Self-Study Guide, ISBN 0-88022-149-6 QUE Corporation, Indianapolis, IN 46250, USA.

Schustack, Steve [1985]

Variations In C, ISBN 0-914845-48-9, Microsoft Press, Box 97017, Redmond, WA 98073-9717, USA.

Schwaderer, W. David [1985]

C Wizard's Programming Reference, ISBN 0-471-82641-3, Wiley Press, New York, NY 10158, USA.

Schildt, Herbert [1987]

C The Complete Reference, ISBN 0-07-881263-1 Osborne/McGraw-Hill, 2600 Tenth Street, Berkely, CA 94710, USA.

Sinclair, Ian [1986]

C For Beginners, ISBN 0-86161-206-X Melbourne House (Publishers) Ltd., 60 High Street, Kingston-Upon-Thames, Surrey KT1 4DB, England.

Sobelman, Gerald E. and David E. Krekelberg [1985]

Advanced C Techniques & Applications, ISBN 0-88022-162-3, QUE Corporation, Indianapolis, IN 46250, USA.

Traister, Robert J. [1985]

Going from BASIC to C, ISBN 0-13-357799-6, Prentice-Hall, Inc., Englewood Cliffs, NJ 07632, USA.

Ward, Robert [1986]

Debugging C, ISBN 0-88022-261-1 QUE Corporation, Indianapolis, IN 46250, USA.

AmigaDOS References

Ariadne Software [1987]

The Kickstart Guide to the Amiga, Ariadne Software Ltd. 273 Kensal Road, London NW10, England.

Berry, John Thomas

Inside the Amiga, ISBN 0-672-22468-2 Howard W.Sams & Co., 4300 W.62nd Street, Indianapolis, IN 46268, USA.

Boom, Michael [1986]

The Amiga: Images, Sounds and Animation ISBN 0-914845-62-4 Microsoft Press, Box 97017, Redmond, WA 98073-9717, USA.

Commodore-Amiga, Inc. [1986]

The AmigaDOS Manual, Bantam Electronic Publishing, 666 Fifth Avenue, New York, NY 10103, USA.

Commodore-Amiga, Inc. [1986]

Amiga ROM Kernal Reference Manual: Exec, ISBN 0-201-11099-7 Addison-Wesley Publishing Company, Reading, MA, USA.

Commodore-Amiga, Inc. [1986]

Amiga ROM Kernel Reference Manual: Libraries and Devices, ISBN 0-201-11078-4 Addison-Wesley Publishing Company, Reading, MA, USA.

Commodore-Amiga, Inc. [1986]

Amiga Intuition Reference Manual, ISBN 0-201-11076-8 Addison-Wesley Publishing Company, Reading, MA, USA.

Commodore-Amiga, Inc. [1986]

Amiga Hardware Reference Manual, ISBN 0-201-11077-6 Addison-Wesley Publishing Company, Reading, MA, USA.

Compute! Publications, Inc. [1986]

Inside Amiga Graphics, ISBN 0-87455-040-8, Compute! Publications, Inc., P.O. Box 5406, Greensboro, NC 27403, USA.

Donald, Bill [1986]

The Amiga System: An Introduction, ISBN 1-85231-001-4, Precision Software Ltd. 6 Park Terrace, Worcester Park, Surrey KT4 7JZ, England.

Mortimore, Eugene P. [1986]

Amiga Programmer's Handbook, Volume I, ISBN 0-895880-343-0, SYBEX Inc., 2021 Challenger Drive #100, Alameda, CA 94501, USA.

Mortimore, Eugene P. [1987]

Amiga Programmer's Handbook, Volume II, ISBN 0-895888-384-8, SYBEX Inc., 2021 Challenger Drive #100, Alameda, CA 94501, USA.

Peck, Robert A. [1987]

Programmer's Guide To The Amiga, ISBN 0-895883-310-4, SYBEX Inc., 2021 Challenger Drive #100, Alameda, CA 94501, USA.

Spanik, Christian [1986]

Presenting The Amiga, ISBN 0-948015-128 First Publishing Ltd., Horseshoe Road, Pangbourne, Berkshire RG8 7SW, England.

Motorola 68xxx References

Barrow, David [1985]

68000 Machine Code Programming, ISBN 0-00-383163-9 William Collins Sons & Co.Ltd., 8 Grafton Street, London W1X 3LA, England.

Eccles, William J. [1985]

Microprocessor Systems A 16-Bit Approach, ISBN 0-201-11985-4 Addison-Wesley Publishing Company, Reading, MA, USA.

Kane, G., D.Hawkins and L.Leventhal [1987]

68000 Assembly Language Programming 2nd Edition, ISBN 0-07-881232-1 Osborne/McGraw-Hill, 2600 Tenth Street, Berkely, CA 94710, USA.

Motorola Inc. [1986]

The MC68000 User's Manual 5th Edition, ISBN 0-13-541475-X Prentice-Hall, Inc., Englewood Cliffs, NJ 07632, USA.

Motorola Inc. [1985]

The MC68020 User's Manual 2nd Edition, ISBN 0-13-541475-X Prentice-Hall, Inc., Englewood Cliffs, NJ 07632, USA.

Motorola Inc. [1985]

The MC68881 User's Manual, Motorola Semiconductor Products Inc., PO Box 20912 Phoenix, AZ 85036, USA.

Robinson, Phillip R. [1985]

Mastering The 68000 Microprocessor, ISBN 0-8306-1886-4 Tab Books Inc., Blue Ridge Summit, PA 17214, USA.

Kelly-Bootle, Stan and Bob Fowler [1985]

68000, 68010, 68020 Primer, ISBN 067-22405-4 Howard W.Sams & Co., 4300 W.62nd Street, Indianapolis, IN 46268, USA.

Williams, Steve [1985]

Programming the 68000, ISBN 0-89588-133-0 SYBEX Inc., 2021 Challenger Drive #100, Alameda, CA 94501, USA.

Whitworth, Ian R. [1985]

16-Bit Microprocessors, ISBN 0-00-383113-2 William Collins Sons & Co.Ltd., 8 Grafton Street, London W1X 3LA, England.

Appendix B. Disk Contents

This appendix lists the contents of the three disks which comprise the Lattice AmigaDOS C Compiler product.

- Disk 1 is a *bootable* system disk containing the executable programs.
- Disk 2 contains the Lattice and Amiga *header* files and libraries in compressed form, together with source files for some of the routines in the Lattice run-time support library.
- Disk 3 contains the Lattice and Amiga *header* files and libraries in uncompressed form, together with a large number of examples.

Disk 1

S (dir)	L (dir)	T (dir)
Devs (dir)	Fonts (dir)	Libs (dir)
System (dir)	System.info	Trashcan (dir)
Trashcan.info	Expansion (dir)	Expansion.info

C (dir) Compiler Programs

asm	blink
fd2pragma	lc
lc1	lc2
lcompact	omd
oml	

Read.me
On Lattice C
On Lattice C.info
setdate

Disk 2

Lib (dir) Compiler run-time support libraries

c.o	cback.o
ovs.o	lc.lib
lcm.lib	lcmffp.lib
lcs.lib	lcms.lib
lcmffps.lib	amiga.lib

CompactH (dir) Compressed (slightly) header files

assert.h	ctype.h
dos.h	errno.h
error.h	fcntl.h
fctype.h	float.h
ios1.h	limits.h
locking.h	math.h
nondp.h	setjmp.h
signal.h	stddef.h
stdio.h	stdlib.h
string.h	time.h
unlstd.h	

clib (dir)
libraries (dir)
workbench (dir)
exec (dir)
graphics (dir)
intuition (dir)
resources (dir)
hardware (dir)
devices (dir)
proto (dir) Prototypes for Amiga functions

Source (dir)

c.a	Startup code
cback.a	Startup code for TSR programs
cxbrk.c	Ctrl-C handler
cxferr.c	Floating point error handler
cxffp.a	FFP low-level interface
cxovf.c	Stack overflow handler - part 1
ffptran.a	FFP transcendental interface
matherr.c	Math library error handler
oserr.c	OS error messages
syserr.c	UNIX equivilant error messages
_assert.c	*'assert'* handler
_cxovf.a	Stack overflow handler - part 2
_main.c	Initialization code

Disk 3

Include (dir) Un-compressed header files

assert.h	ctype.h
dos.h	errno.h
error.h	fcntl.h
fctype.h	float.h
ios1.h	limits.h
locking.h	math.h
nondp.h	setjmp.h
signal.h	stddef.h
stdio.h	stdlib.h
string.h	time.h
unlstd.h	

clib (dir)
libraries (dir)
workbench (dir)
exec (dir)
graphics (dir)
intuition (dir)
resources (dir)
hardware (dir)
devices (dir)
proto (dir) Prototypes for Amiga functions

Examples (dir)

boxomatic.c	cat.c
ftoc.c	lines.c
memsup.a	memwatch
memwatch.c	memwatch.doc
popcli	popcli.c
popcli.doc	popsup.a
setdate.c	speechtoy.c

NOTE: The *Include* file directories also contain the header files for the 68000 assembler. These are recognizable by the file extension of *.i* rather than the file extension of *.h* for the C header files.

Appendix C. Commands

Appendix C describes the AmigaDOS commands that are provided with the Lattice C Compiler or that must be used in conjunction with the compiler.

NAME

asm 68000 Macro Assembler

SYNOPSIS

```
asm >listfile options filename
```

DESCRIPTION

The AmigaDOS 68000 macro assembler is designed to generate assembly language components for inclusion in C programs, but it can also be used to generate programs entirely written in assembly language. It supports the full set of 68000 mnemonics, as well as an extensive set of assembler directives and a powerful macro facility.

The assembler reads an assembly language source file and produces an object file in the Amiga object file format, along with an optional listing of the source and assembled code. The format of the command to invoke the assembler is:

```
asm [>listfile] [options] filename
```

The various command line specifiers are shown in the order they must appear in the command. Optional specifiers are shown enclosed in brackets.

>listfile — Causes the listing and error message output of the assembler to be directed to the specified file.

options — Assembler options are specified as a hyphen followed by a single letter; in some cases, additional text may be appended. The letter may be supplied in either upper or lower case. Each option must be specified separately, with a separate hyphen and letter (that is, they cannot be combined as they can for certain UNIX programs). Current options include:

-c — This option specifies that the sections designated by the characters which

immediately follow are to be loaded into memory addressable by the Amiga's custom chips. This is necessary for screen image data. The **-c** option must be immediately followed by one or more of the following letters in any order:

b bss or uninitialized data

c code segment

d data segment

The default action is to load the segments into memory not addressable by the custom hardware if it is available. This is generally desirable because it avoids bus contention between the processor and the custom hardware. Image and audio data must be loaded into memory accessible to the custom hardware. For example:

```
-acdb
```

will cause all segments to be loaded into memory addressable by the custom hardware, regardless of the system memory configuration.

Specifies that INCLUDE files are to be searched for by prepending the filename with the string prefix, unless the filename in the INCLUDE statement is already prefixed by a drive or directory specifier. Up to 4 different **-i** strings may be specified in the same command. When an unprefixed INCLUDE filename is encountered, the current directory is searched first; then file names are

constructed and searched for, using prefixes specified in **-i** options, in the same left-to-right order as they were supplied on the command line. No intervening blanks are permitted in the string following the **-i**. Note that if a directory name is to be specified as a prefix, a trailing slash must be supplied.

-l[list] Causes a listing of the source file to be written to the standard output. The listing displays the appropriate location counter and data generation information alongside the assembly source. One or more of the following characters may be appended to the **-l** option, with the following effects:

x Lists the expansion text for macros.

m Lists additional data generated for source lines which cannot be accommodated alongside the original source line (i.e. allows multiple listing lines for each source line).

i Lists the source for text from INCLUDE files as well as the original source file.

-oprefix Specifies that the output filename (the .o file) is to be formed by prepending the input filename (the .a file which is being assembled) with **prefix**. Any drive or directory prefixes originally attached to the input filename are discarded before the new prefix is added. No intervening blanks are permitted in the string following the **-o**. Note that if a directory name is to be specified as a prefix, a trailing slash must be supplied.

filename — Specifies the name of the assembly language source file which is to be assembled; this is the only command line field which must be present. If the filename is specified without an extension, .a is assumed. The object file created by the assembler will have the same name as the source file, except that .o will be supplied in place of any extension.

EXAMPLES

This command causes the assembly language source file **modn.a** to be assembled, producing the object file **modn.o**. A listing of the source file, along with any error messages generated, will be written to the file **modn.lst**.

```
asm >modn.lst -l modn
```

NAME

assign logical name assignment

SYNOPSIS

```
assign logical_name path
```

DESCRIPTION

The AmigaDOS **assign** command is used to establish a relationship between an arbitrary name and a disk directory or file. When the assignment of a name to a directory or file is made, you can use the logical name in place of the directory or file specification. For example:

```
assign myfiles: mydisk:project/source
```

relates the logical name **myfiles:** to the directory **project/source** on the disk volume **mydisk:**. After this assignment is made, the file specification:

```
myfiles:somefile.c
```

would be synonomous with:

```
mydisk:project/source/somefile.c
```

Logical name assignments may be removed by making the assignment without the second parameter. For example:

```
assign mydisk:
```

removes any previous assignment of the logical name **mydisk:**.

The Lattice AmigaDOS compiler uses various logical names to help locate files required to compile and link C programs.

SEE ALSO

lc

NAME

lc Lattice C Compiler

SYNOPSIS

```
assign LC: executable path
assign INCLUDE: include path
assign LIB: library path
assign QUAD: quad path

lc options files
```

DESCRIPTION

The normal method for executing the Lattice C Compiler is to invoke the **lc** command. This program separates the **options** list into those for pass 1 and those for pass 2. Then it executes **lc1** (pass 1) and **lc2** (pass 2) for each of the C source files specified by the **files** list. This list can consist of one or more file names and/or file patterns, separated by white space. For example:

```
lc #? :mydir/myprog :mydir/abc?
```

will compile all C source files in the current directory, plus the source file named **:mydir/myprog.c** plus all C source programs in the **:mydir** directory which have four-character names beginning with *abc*. Note that the **lc** command automatically supplies the *.C* extension on all source file names.

The LC command will also automatically invoke the librarian and linker if required.

This command utilizes the logical name assignment feature of AmigaDOS to locate the various programs and files. These assignments allow these programs and files to be located in any directory on any disk.

The **options** list can contain any combination of the following, separated by blanks:

-a This option specifies that the sections designated by the characters which immediately follow are to be loaded into memory addressable by the Amiga's custom hardware. This is necessary for screen image and audio data. This option is synonymous with the **-c** option for **lc2**. The **-a** option must be immediately followed by one or more of the following letters in any order:

b bss or uninitialized data

c code segment

d data segment

The default action is to load the segments into memory not addressable by the custom hardware if it is available. This is generally desirable because it avoids bus contention between the processor and the custom hardware. Image and audio data must be loaded into memory which accessible by the custom hardware. For example:

```
-acdb
```

will cause all segments to be loaded into chip addressable memory, regardless of the system memory configuration.

-b This option causes the compiler to use a base-relative form of addressing when referring to data section locations. This is a more compact instruction format that uses a 16-bit offset field instead of a 32-bit absolute address. The data is addressed relative to the contents of address register A4. This method has the disadvantage of only allowing 64K bytes of data that can be addressed using this option, and all the data addressed this way must be in the same hunk. The default is to use 32-bit absolute addressing, which allows data hunks of virtually unlimited size that can reside anywhere in memory.

This option cannot be used with programs that will be used as interrupt handlers or that will be used with the **AddTask()** function unless the **-y** option is also used.

-C Note that the key character is an upper case C and not a lower case character. This option causes the LC command to continue with the next source file when a fatal compilation error is reported while multiple source files are being compiled. Normally, a fatal error causes the process to pause with the following message displayed on your screen:

```
Compiler return code xx.
Press Y to abort, any other key to continue.
```

The compiler error messages are also displayed immediately above this prompter. If you respond with a 'Y' (yes), LC will abort, otherwise it will proceed to the next source file.

-c Note that the key character is an lower case C and not an upper case character. The compiler defaults to the ANSI C language standard, but the **-c** *compatibility option* can be used to activate some deviations from the standard, and in this way achieve compatibility with previous versions of Lattice C. The **-c** must be immediately followed by one or more letters from the following list, in any order:

c Allows comments to be nested.

d Allows $ character to be used in identifiers.

m Allows use of multiple character constants (e.g. *ab*).

s Causes compiler to generate a single copy of identical string constants.

t Enables warning messages for structure and union tags that are used without being defined. For example:

```
struct XYZ *p;
```

would not normally produce a warning message if the structure tag XYZ were not defined.

u Forces all *char* declarations to be treated as *unsigned char.*

w Shuts off warning messages generated for *return* statements which do not specify a return value within an *int* function. By the ANSI standard, such functions should be declared as *void* instead of *int.*

-d This option has two uses. When there is no variable field following the **-d**, it activates the debugging mode. Additional information is placed into the object module to facilitate symbolic debugging. Also, the preprocessor symbol **DEBUG** is defined so any debugging statements in the source file will be compiled.

The **-d** option can also be used to define preprocessor symbols in the following ways:

-dsymbol Causes **symbol** to be defined as if you had included the statement

```
#define symbol
```

in your source file.

-dsymbol=value Causes **symbol** to be defined as if you had included this statement in your source file:

```
#define symbol value
```

-e This option causes the compiler to recognize the extended character set used in Asian-language applications. Note that this option has no validity on the current generation of hardware running v1.2 of the system software.

-f This option causes the compiler to use the Motorola Fast Floating Point format for all floating point operations. The resulting object module must be linked with the FFP math library **lcmffp.lib**. Normally the compiler uses the IEEE standard for floating point representation, which is compatible with the MC68881 floating point coprocessor. Files compiled with this option are not compatible with files which use the IEEE format and may not be linked with object modules which use the IEEE format.

-h This option specifies that the sections designated by the characters which immediately follow are to be loaded into memory not addressable by the Amiga's custom hardware. The **-h** option must be immediately followed by one or more of the following letters in any order:

b bss or uninitialized data

c code segment

d data segment

The default action is to load the segments into memory not addressable by the custom hardware if possible. This option prevents the specified segments from being loaded into chip addressable memory even if no other memory is available. This can cause programs to not execute if external high-speed memory is not available on the machine. As an example:

```
-hcdb
```

would cause all three sections to be loaded only in high-speed ram.

-i This option specifies a directory that the compiler should search when it is attempting to find an include file that is enclosed in double quotes. For example, if you specify the option as **-idf0:headers/ -idf1:local/** and then place the line:

```
#include "defs.h"
```

in your source program, the compiler first tries to find the header file named **defs.h** in the current directory. If it is not there, then the compiler searches for **df0:headers/defs.h** and **df1:local/defs.h** in this order. Finally, if these attempts fail, the compiler continues the search using the directories specified in the INCLUDE logical name assignment.

Note that you can place up to ten **-i** options on the command line.

-L Note that the key character is an upper case L and not a lower case character. When this option is present, **lc** invokes the linker if all compilations are successful. The first source file name is used as the name of the executable and map files produced by the linker. Any other files that were compiled are supplied to the linker as secondary object files. The Lattice C startup routine is included as the first object module.

The linker is instructed to search the standard Lattice library file **lc.lib** and the standard Amiga library file **amiga.lib**.

Additional Lattice libraries and linker options may be specified by immediately following the **-L** option with one or more of the following letters:

c This invokes the SMALLCODE option of the linker. It causes all code hunks to be combined into a single code hunk. **-b** option.

d This invokes the SMALLDATA option of the linker. It causes all data and bss hunks to be combined into the single data hunk that is referenced by modules compiled with the **-b** option.

f This letter specifies that the Motorola FFP math library **lcmffp.lib** is to be searched before the standard run-time support library.

h This letter directs the linker to output the hunk portion of the map. This is the default map if no other map options are specified.

l This letter directs the linker to include library information in the map file.

m This letter specifies that the Lattice IEEE math library **lcm.lib** is to be searched before the standard run-time support library.

o This directs the linker to include overlay information in the map file.

s This letter directs the linker to produce a symbol listing in the map file.

v This invokes the VERBOSE option of the linker. It causes the linker to display statistical messages as it is processing the object files and libraries.

For example, **-Lm** will search **lcm.lib** before **lc.lib** and **amiga.lib**, and **-Lvf** will search **lcmffp.lib, lc.lib**, and **amiga.lib**, and display messages regarding the current linker status. Note that the math library options are mutually exclusive, and that the standard libraries are always searched last.

If you want to search other libraries, you must list those libraries after the letters options, and use plus signs as separators. For example, **-L+myfuncs.lib** searches **myfuncs.lib** before the standard Lattice and Amiga libraries, while **-Lm+myfuncs.lib+:george/myfuncs.lib** searches the libraries **myfuncs.lib, :george/myfuncs.lib, lcm.lib, lc.lib**, and **amiga.lib**. Note that the special libraries are searched before the Lattice libraries.

The **-L** option creates a file in the current directory named **xxx.lnk**, where **xxx** is the name of the first source file to be compiled (i.e., the same name that is used for the executable and map files). This *.LNK* file serves as input to the linker, and it is not deleted at the end of the procedure. This allows you to easily re-link if, during your testing, you find a need to change and re-compile only one module. To do this, simply execute the linker utility **blink** in the following way:

```
blink WITH xxx.lnk
```

where **xxx.lnk** is the name of the .LNK file previously produced by the LC command.

-l Note that the key character is an lower case L and not an upper case character. This option causes all objects except characters, short integers, and structures that contain only characters and short integers to be aligned on longword boundaries (i.e., addresses evenly divisible by 4).

Structures will be longword aligned if they contain any members that must be aligned.

-M When this option is present, LC will compile only those source files with dates more recent than the corresponding object files. Note that the dates of included files are not checked. In other words, if you change a header file without changing the source file that includes it, the source file will not automatically re-compile because it still pre-dates its object file.

-n This option causes the compiler to retain only 8 characters for all identifiers. The default maximum identifier length is 31 characters. In either case, anything beyond the maximum length is ignored.

-o This option should be followed by the drive, directory, or complete file name for the object file that is produced by pass 2. Several examples are:

-odf0: Places the object file in the root directory on drive df0:.

-o:obj/ Places the object file into directory **obj/** on the current drive. The name of the file is the same as the source file name, with a .O extension instead of .C.

-ospecial.o Places the object file into the current directory with the name **special.o**.

-q This option should be followed by the drive, directory, or complete file name for the *quad file*, which is the intermediate file generated by pass 1 and read by pass 2. Several examples are:

-qram: Places the quad file in ram disk.

-qdf0: Places the quad file in the root directory of drive df0:.

-q:quads/ Places the quad file into directory **quad/** on the current drive. The name of the file is the same as

the source file name, with a .Q extension instead of .C.

-qspecial.qqq Places the quad file into the current directory with the name **special.qqq**.

Note that the quad file is automatically deleted after pass 1. If you do not want this to happen, call the **lc1** command directly.

-r Note that the key character is an lower case R and not an upper case character. This option is used to cause the compiler to use pc-relative addresses instead of absolute addresses for external function calls. Target locations must be within +/-32K of the instruction. For small programs this form of branching can be very efficient. However, in larger programs the target function may not be within the necessary range. If the linker being used does not automatically adjust these instructions then such programs may not be linkable. If the linker does support the automatic adjustments necessary, large programs may run slower because out-of-range branches must be routed through a jump table. In general, however, this option can produce more compact load modules.

-R Note that the key character is an upper case R and not a lower case character. When this option is specified, the object modules produced by the compiler are automatically inserted into a library file, replacing modules of the same names. The option must be followed by the name of the library, as in

```
-Rmylib.lib
```

which places the object modules into the **mylib.lib** library file. The **-R** can be followed by any valid file name, including drive code and path. A .lib extension is not automatically supplied.

-s This option allows you to change the default hunk or segment names generated by the compiler. If it is not present, the hunks are unnamed. You can use this option to provide hunk names as follows:

-s — This causes the compiler to use the default names of **text** for the program section, **data** for the data section, and **udata** for the bss or uninitialized data section.

-sc=codename — Causes the compiler to use the name **codename** for the program, or code, section without affecting the names of the other sections.

-sd=dataname — Causes the compiler to use the name **dataname** for the data section without affecting the names of the other sections.

-sb=bssname — Causes the compiler to use the name **bssname** for the bss, or uninitialized data, section without affecting the names of the other sections.

-u — This option by itself undefines all preprocessor symbols which are normally pre-defined by the compiler. If you want to undefine a specific symbol, it must immediately follow the **-u**. For example, **-uAMIGA** undefines the AMIGA symbol.

-v — This option disables the generation of stack checking code at the beginning of each function.

-x — This option causes all global data declarations to be treated as externals. This can be useful if you define data in a header file that is included by multiple source files. The **-x** option can be used with all the files except one, in this case, to cause the data items to be defined in one module and referenced as externals in the others.

-y — This option causes each function entry sequence to load address register A6 with the value of the linker defined symbol **LinkerDB**. This symbol is the data section base address, biased as necessary. This option must be used if the **-b** option is used with interrupt code or functions that will be multi-tasked with the **AddTask()** function. Note that in general only the functions that can be used as entry points to the task or interrupt handler need to use this feature, since register A4 will be propagated by subsequent function calls.

RETURNS

The LC command returns the following completion codes:

0 All compilations were successful. That is, at least one source program was compiled, and there were no fatal errors.

1 One or more fatal compilation errors were reported.

2 No source files were found.

SEE ALSO

lc1, lc2, link

EXAMPLES

The first command assigns the INCLUDE logical name so that header files will be found in the **df0:include** directory. The second command assigns the LC logical name so that the driver program can find the executables in the **df0:c** directory. The next command assigns the LIB logical name so that the driver program can tell the linker to locate the standard library files in the **df0:lib** directory. The next command compiles all modules in the **mylib** subdirectory and then constructs a library named **mylib.lib** if all compilations are successful. The last command compiles **myprog.c** and links it with **mylib.lib** to produce a load module named **myprog**.

```
assign INCLUDE: df0:include
assign LC: df0:c
assign LIB: df0:lib
lc  -Rmylib mylib/#?
lc -mp -L+mylib myprog
```

NAME

lc1 Compiler pass 1

SYNOPSIS

```
assign INCLUDE: include path

lc1 options file
```

DESCRIPTION

This command invokes the first compiler pass, which reads a source file and translates it into an intermediate form known as a *quad file*. Unlike the **lc** command, you can only specify one source file to **lc1**, and it should be written without the .C extension. For example, if the file argument is **myprog**, this pass will translate **myprog.c** into the quad file named **myprog.q**.

The **options** argument can consist of the following items, which are described in the **lc** command:

-b Base register relative data addressing. Note that this now the default. To switch this mode off use **-b0** option. The options **-b** or **-b1** will switch base register relative data addressing back on again.

-c Compiler compatibility settings.

-d Debugging mode or preprocessor symbol definition.

-e Extended character set processing.

-f Motorola Fast Floating Point format.

-i Directory paths for local include files.

-l Longword alignment of data items.

-n Long symbol names.

-o Specify destination for .Q file. Note that this option is specified as **-q** on the **lc** command.

-p Preprocess only. This option is not available with the **lc** command. When this option is specified, **lc1** only performs the preprocessor phase and writes the output to a file of the same name with a .p extension.

-u Undefine preprocessor symbols.

-w Generate code to use short integers.

All options must appear before the file name.

SEE ALSO

lc, lc2

NAME

lc2 Compiler pass 2

SYNOPSIS

```
lc2 options file
```

DESCRIPTION

This command invokes the second compiler pass, which reads a quad file and translates it into an object file. Note that you can only specify one quad file to **lc2**, and it should be written without the .Q extension. For example, if the file argument is **myprog**, this pass will translate **myprog.q** into **myprog.o**.

The **options** argument can consist of the following items, which are described in the **lc** command:

-c	Chip memory specification. This is the same as the **-a** option on the **lc** command.
-h	Fast memory specification.
-o	Specify destination for .O file.
-r	Use pc-relative subroutine calls. This is now the default. To switch it off, use the **-r0** option. To return to the default condition, use the **-r** or **-r0** options.
-s	Specify segment name.
-v	Disable stack checking.
-y	Unconditionally load the base register

All options must appear before the file name.

SEE ALSO

lc, lc1

NAME

omd Object Module Disassembler

SYNOPSIS

```
omd >output options object source
```

DESCRIPTION

This utility program disassembles an object file produced by the Lattice C Compiler and produces an output listing consisting of assembly-language statements, possibly interspersed with the original C source code.

The **object** field is required and gives the complete object file name. That is, OMD will not automatically supply the .o extension. The **source** field is optional. If present, it must be the complete source file name. When this field is used, you should have compiled the source file with the **-d** option (see the **lc** command) to produce the debugging information in the object module that allows **omd** to associate a particular source line with the object code that was generated. If you did not use the **-d** option, then the C source lines will not appear in **omd's** output.

The **>output** field is optional and redirects **omd's** output from the screen to an output device or file. Most programmers use **omd** by redirecting its output to a file and then printing the file. See the example below.

The **options** field need not be present. The Amiga implementation of **omd** only accepts the following option:

-x This option overrides the default size of the buffer used to hold the external symbol section of the object module. For example, **-x200** establishes a buffer that can hold 200 external symbols, which is the default. You should increase this value if **omd** reports that there are too many external symbols.

EXAMPLES

This example compiles *MYPROG.C* to produce *MYPROG.O*, which is then disassembled. The disassembled listing is saved in the file *MYPROG.LST*.

```
lc -d  myprog
omd >myprog.lst myprog.o myprog.c
```

NAME

oml Object Module Librarian

SYNOPSIS

```
oml [<cmdfile] [>listfile] [options] libfile [commands]
```

DESCRIPTION

The object module librarian **oml** can create a library file by combining object modules, generate a listing of the modules (and their public symbols) contained in a library, or manipulate modules within an existing library file.

Library files provide a convenient way of collecting object modules to be presented as a group of available components during program linking; the load module builder then includes only those modules from the library which are actually needed by the load module being built. Libraries are especially useful when several programs make use of common subroutines, since these subroutines can be placed in a library file and included on an *as required* basis when the programs are linked.

A library file is made up of object modules, each of which was originally a single file. Each module within the library is identified by a module name, which is normally obtained from the object module itself (the Amiga object module format defines a special *program unit* or module name record). This name is placed in the object module by the translator (assembler or compiler) program which generates it. Some modules may not contain a module name at all; in that case, the object module librarian assigns a module name of the form **$nnn**, where **nnn** is a decimal number indicating that the module was the **nth** unnamed module encountered in the library.

In order to perform replacement of modules within a library file, it is necessary to insure that the module contains a program unit or module name. The Lattice AmigaDOS C Compiler and AmigaDOS assembler always place a module name in the object files they produce. The current versions of the compiler and assembler use the name of the object file. Thus, compiling **ftoc.c** produces an object module with the name **ftoc.o**.

The Metacomco Amiga assembler does not place any module name in the output file unless the IDNT directive is used. For example:

```
IDNT    "asmod.o"
```

causes the module name **asmod.o** to be placed in the object file created when the source file is assembled.

When the linker examines a library file to find modules to be incorporated into a program, the module name is not important; instead, the linker decides if a module is needed on the basis of the public symbols it defines. A module may define one or more such symbols, which identify program components such as functions or data elements. Because the presence of more than one definition for a symbol may cause confusion, the object module librarian warns when it examines or constructs a library file which includes multiple definitions of a symbol.

Each invocation of **oml** specifies a particular library file upon which operations are to be performed. Then, a sequence of one or more commands is used to indicate the desired operations.

Commands may be specified on the command line used to execute **oml**, or they may be read from **stdin**, or a combination of both kinds of input may be used. If no commands are specified on the command line which executed **oml**, they are automatically read from **stdin**, which is usually the user's console but can be redirected to a file. The special command @ (valid only on the command line) is used to switch command input from the command line to **stdin**; an explicit file name may be attached to the @ to force commands to be read from that file. Commands are read from a file or from **stdin** until an end of file condition is detected; if commands are being read from the user's console, a control-backslash must be used to end command input.

Each command is specified as a single character, usually followed by a list of module names or object file names. Commands and file/module names are separated from each other by white space. If a command is followed by one or more names, the first name specified is NOT checked as a possible command; thus, names which might be confused with commands must be specified as the FIRST name following a command.

The format of the command to invoke the object module librarian is:

```
oml [<cmdfile] [>listfile] [options] libfile [commands]
```

The various command line specifiers are shown in the order they must appear in the command. Optional specifiers are shown enclosed in brackets.

<cmdfile — Causes commands to be read from the named file, provided that (1) no commands are specified on the command line or (2) the @ command (see below) is used to force commands to be read from **stdin.** If this option is omitted and neither of the above conditions is met, commands are read from the user's console.

>mapfile — Causes the listing output generated by **oml** to be written to the named file. If omitted, listing output is directed to the console.

options — Librarian options are specified as a hyphen followed by a string of characters which may not include white space. Current options include the following:

-oprefix — Specifies that the output filenames for the **x** command are to be formed by prepending the module name with **prefix.** Note that if a directory name is to be specified as a prefix, a trailing node separator (a slash under AmigaDOS) must be supplied on the prefix.

-s — Causes a listing of the public symbols defined in the module to be included in the listing produced by the **l** command.

libfile — Specifies the name of the library file to be created or manipulated; this is the only command line field which must be present.

commands — These specify the actions to be performed by **oml** with respect to the specified library file. Commands are specified by a single character; if alphabetic, either upper or lower case may be used. They are separated from each other and from elements of file or module name lists by any white space. Current commands include:

r file-list	Replaces the named object files in the library, or adds them to the library if not already present. Note that replacement of existing modules in a library will work correctly only if the file name is the same as the module name. If the module does not exist in the library, this is unnecessary.
d module-list	Deletes the named modules from the library. Since modules without program unit names are assigned module names by the librarian, it may be necessary to obtain a listing (via the **l** command) in order to determine the assigned **$nnn** name for the module which is to be deleted.
x module-list	Extracts the named modules from the library, creating files of the same names. Note that if the module name includes a path name, the librarian will attempt to create a file with that name. All files are created in the current directory, unless the *-o* option is used; in that case, each module name is prefixed with the text specified on the *-o* option. The special character '*' may be used to indicate that all of the modules in the library are to be extracted. **oml** will terminate execution if an attempt to extract a module is unsuccessful. Note that it is an error to specify the same name in both a replacement and an extraction list.
l	Causes generation of a listing of the modules in the library after all other requested operations have been performed. If the **-s** option is used on the command line which invokes **oml**, the listing will include the public symbols defined in each module as well as a list of the module names themselves.

@[filename]	Causes the remainder of command input to be read from **stdin**, or from the named file. Note: brackets are used to show that the file name is optional, and are not included if a file name is specified).

If replacement modules or deletions are specified, a new version of the library file will be built, provided that no errors are detected. This new version is created first as a temporary file; when it has been completely built, the original library file (if it existed) is deleted, and the temporary file renamed. This sequence insures that the original library file will not be affected if an error is detected. Modules are always included in a library in topologically sorted order, so that no backward references occur (except in the case of modules which reference each other, which are retained in the same order in which they are encountered).

Warning messages are produced if:

- A module named in a deletion or extraction list was not found in the library.
- If a second definition for a public symbol is encountered in one of the modules to be included.

EXAMPLES

The following examples illustrate the use of **oml**. Remember that replacing modules within a library file only works correctly if the module name is the same as the file name. Since a single object file is essentially a library of one module, **oml** can be used to find out what module name is included in the object file by using the following command:

```
oml name.o l
```

Building a New Library

Create a list of the file names of the object modules which will make up the library. The Lattice **extract** utility can be used to accomplish this. Then create the library using the following command:

```
oml new.lib r @name.lst
```

where **new.lib** is the name of the library to be created, and **name.lst** contains a list of the files to be included in the library. Note that creation of a new library is one occasion where the correspondence between file and module names is not required.

Extracting Modules from a Library

Use the following command to break out all of the modules from a library:

```
oml -o:object/ cfuncs.lib x *
```

Note that this command will be successful only if no module names in the library CFUNCS.L contain path names. A file for each of the modules in the library will be created in the directory **:object/** in this example.

Deleting Modules from a Library

Use the following command to delete modules from a library:

```
oml cfuncs.lib d tribe.o
```

This example deletes the module **tribe.o** from the library **cfuncs.lib**.

Listing the Modules in a Library

Use the following command to obtain a listing of the modules and symbols in a library file:

```
oml -s test.lib l
```

The listing may be saved to a file using I/O redirection:

```
oml >test.lst -s test.lib l
```

SEE ALSO

blink,lc

Appendix D. Data Names

Appendix D describes the public data names that are defined in the Lattice Libraries.

DOSBase

AmigaDOS Library Vector *Class: AMIGA*

NAME

DOSBase AmigaDOS Library Vector

SYNOPSIS

```
extern long DOSBase;
DOSBase = OpenLibrary("dos.library",ver);
```

DESCRIPTION

This external location is used by various Amiga library routines which interface with AmigaDOS system functions. It is initialized by an **OpenLibrary** call in the startup code and is expected to contain the base address of the AmigaDOS system library vector table. If you do not link with the startup module **c.o** and you make calls to AmigaDOS system functions or use Lattice features that call AmigaDOS system functions, then you must first initialize this location by calling **OpenLibrary**.

NAME

errno	UNIX error number
sys_nerr	Number of UNIX error codes
sys_errlist	UNIX error messages

SYNOPSIS

```
#include <error.h>
extern int errno;
extern int sys_nerr;
extern char *sys_errlist[ ];
```

DESCRIPTION

The external integer named **errno** is initialized to 0 at start-up time. Then if an error is detected by one of the standard library functions, a non-zero value is placed there. The standard library never resets **errno**.

Programmers typically use this information in two ways. In some cases, it is appropriate to check **errno** after a sequence of operations and abort if any error occurred along the way. In other cases, **errno** is checked periodically, and if it is non-zero, the appropriate corrective action is taken. Then the application program resets **errno** before beginning the next processing phase.

The **sys_nerr** and **sys_errlist** items are defined in a C source file named **syserr.c** and are used by the **perror** function to print messages that correspond to the code found in **errno**.

Note that even though error information is normally placed into **errno** by the standard library functions, application programs can also use this technique to indicate problems. However, you should be careful about adding new codes and messages just above the highest UNIX code currently defined, since new UNIX codes are occasionally added. Also, we recommend that you add application-dependent codes by extending the

header file **error.h**, which contains symbolic definitions of the code numbers. The currently-defined codes are listed below:

Symbol	Code	Meaning
EOSERR	-1	Operating system error
EPERM	01	User is not owner
ENOENT	02	No such file or directory
ESRCH	03	No such process
EINTR	04	Interrupted system call
EIO	05	I/O error
ENXIO	06	No such device or address
E2BIG	07	Arg list is too long
ENOEXEC	08	Exec format error
EBADF	09	Bad file number
ECHILD	10	No child process
EAGAIN	11	No more processes allowed
ENOMEM	12	No memory available
EACCES	13	Access denied
EFAULT	14	Bad address
ENOTBLK	15	Bulk device required
EBUSY	16	Resource is busy
EEXIST	17	File already exists
EXDEV	18	Cross-device link
ENODEV	19	No such device
ENOTDIR	20	Is not a directory
EISDIR	21	Is a directory
EINVAL	22	Invalid argument
ENFILE	23	No more files (system)
EMFILE	24	No more files (process)
ENOTTY	25	Not a terminal
ETXTBSY	26	Text file is busy

EFBIG	27	File is too large
ENOSPC	28	No space left
ESPIPE	29	Seek issued to pipe
EROFS	30	Read-only file system
EMLINK	31	Too many links
EPIPE	32	Broken pipe
EDOM	33	Math function argument error
ERANGE	34	Math function result is out of range

SEE ALSO

perror

DiskfontBase

Disk Font Library Vector *Class: AMIGA*

NAME

DiskfontBase Disk Font Library Vector

SYNOPSIS

```
extern long DiskfontBase;
DiskfontBase = OpenLibrary("diskfont.library",ver);
```

DESCRIPTION

This external location is used by various Amiga library routines which interface with the ROM Kernel text functions which deal with new fonts. It must be initialized by an **OpenLibrary** call before any of the disk font functions documented in the Amiga ROM Kernel manuals can be called. It is expected to contain the base address of the disk font library vector table.

NAME

GfxBase Graphics Library Vector

SYNOPSIS

```
extern long GfxBase;
GfxBase = OpenLibrary("graphics.library",ver);
```

DESCRIPTION

This external location is used by various Amiga library routines which interface with the ROM Kernel graphics system functions. It must be initialized by an **OpenLibrary** call before any of the graphics functions documented in the Amiga ROM Kernel manuals can be called. It is expected to contain the base address of the graphics library vector table.

NAME

IntuitionBase Intuition Library Vector

SYNOPSIS

```
extern long IntuitionBase;
```

DESCRIPTION

This external location is used by various Amiga library routines that interface with the Intuition system functions. It must be initialized by an **OpenLibrary** call before any of the functions documented in the Amiga Intuition manuals can be called. It is expected to contain the base address of the Intuition library vector table.

NAME

MathBase FFP Library Vector

SYNOPSIS

```
extern long MathBase;
MathBase = OpenLibrary("mathffp.library",ver);
```

DESCRIPTION

This external location is used to interface with the Motorola Fast Floating Point library routines provided by Amiga. It is initialized by an **OpenLibrary** call when a program compiled with the FFP option performs the first floating point operation. It contains the base address of the FFP math library vector table. If you make direct calls to the Amiga FFP functions you must first initialize this location by calling **OpenLibrary**.

NAME

msflag MS-DOS File Pattern Flag

SYNOPSIS

```
extern int msflag;
```

DESCRIPTION

This external integer is used by the file name pattern matching functions to specify AmigaDOS or MSDOS wildcard characters. If **msflag** is non-zero then MSDOS file name patterns are used. The default is to use AmigaDOS file name patterns.

SEE ALSO

dfind, getfnl

NAME

MathTranBase FFP TRig Library Vector

SYNOPSIS

```
extern long MathTransBase;
MathTransBase = OpenLibrary("mathtrans.library",ver);
```

DESCRIPTION

This external location is used to interface with the Motorola Fast Floating Point format transcendental function library routines provided by Amiga. It is initialized by an **OpenLibrary** call when a program compiled with the FFP option performs the first floating point transcendental function call. It contains the base address of the FFP transcendental math library vector table. If you make direct calls to the Amiga FFP transcendental functions you must first initialize this location by calling **OpenLibrary**.

NAME

_timedata	Time Zone variables
daylight	Daylight savings time flag
timezone	Timezone bias from GMT
tzname	Timezone names
tzstn	Standard time name
tzdtn	Daylight time name

SYNOPSIS

```
extern int daylight;
extern long timezone;
extern char *tzname[2];
extern char tzstn[4];
extern char tzdtn[4];
```

DESCRIPTION

These variables are initialized by the **tzset** function and are used by the **localtime** function to adjust from Greenwich Mean Time to the local time.

The **daylight** item is non-zero if daylight saving time is currently in effect. The **timezone** value is the number of seconds that must be subtracted from GMT. The two **tzname** pointers point to **tzstn** and **tzdtn**, respectively. These strings contain the three-character names for standard time (**tzstn**) and daylight time (**tzdtn**).

SEE ALSO

localtime, tzset

NAME

_fmode Default level 2 I/O mode

SYNOPSIS

```
extern int _fmode;
```

DESCRIPTION

This external integer is used by the **fopen** function to determine the translation mode to use when the programmer does not specify a mode in the **fopen** call. For AmigaDOS it is set to 0x8000, which specifies untranslated mode.

SEE ALSO

fopen

NAME

_FPERR Floating Point Error Code

SYNOPSIS

```
extern int _FPERR;
```

DESCRIPTION

This location will contain a non-zero value after any low-level floating point operation encounters an error. Low-level operations include addition, subtraction, multiplication, division, comparison, and conversion from one number representation to another (e.g. float to double).

The error codes and their corresponding symbols from **math.h** and **math.mac** are:

SYMBOL	VALUE	MEANING
FPEUND	1	Underflow
FPEOVF	2	Overflow
FPEDVZ	3	Divide by zero
FPENAN	4	Not a valid number
FPECOM	5	Not comparable

When the error occurs, the low-level operation passes the appropriate error code to **CXFERR**, which must store the code in **_FPERR**. Note that **_FPERR** is never reset by any low-level operation.

SEE ALSO

CXFERR, _FPA

EXAMPLES

```
/*
*
* This example performs uses the division operation
* to stimulate floating point errors.
*
**/
#include <math.h>
#include <stdio.h>
extern int _FPERR;
main()

double a,b,c;

while(feof(stdin) == 0)

  printf("Enter divisor:   ");
  if(scanf("%lf",&a) != 1) exit(0);
  printf("Enter dividend: ");
  if(scanf("%lf",&b) != 1) exit(0);
  _FPERR = 0;
  c = b / a;
  printf("_FPERR = %d\n",_FPERR);
  printf("%e / %e = %e\n\n",b,a,c);
```

NAME

_MNEED Minimum Dynamic Memory Needed

SYNOPSIS

```
extern long _MNEED; to reference default value
long _MNEED = n; to set initial value
```

DESCRIPTION

_MNEED indicates the minimum number of bytes that must be provided as the so-called *heap space* for use by the various memory allocators. This space, whose size is rounded up to be an even multiple of **_mstep**, is the initial block of memory in the local memory pool.

If one of the memory allocators later determines that it needs more heap space, it will call upon **sbrk** to request additional memory from AmigaDOS. This memory then becomes a non-contiguous addition to the local pool.

Each time **rbrk** is called, all the memory in the local pool is returned to the system, and a request for the amount of memory in **_MNEED** is made of AmigaDOS.

Because of this technique, you can use **_MNEED** to guarantee at the outset that your program will have enough heap space. However, if you don't do anything with **_MNEED**, the memory allocators will still try to honor your space requests until AmigaDOS indicates that no more memory is available.

There are two ways to initialize this item. If your space needs are known at compilation time, you can simply declare **_MNEED** as an initialized public symbol. That declaration will then override the default provided by the library, and the space will be obtained automatically by the startup routine (c.o). However, if you cannot determine your space needs until run time, you can load the appropriate value into **_MNEED** and then call **rbrk**. This, of course, should be only be done after all dynamic memory has been released via the appropriate de-allocator function. See the **rbrk** description for further information and cautions.

NAME

_mstep Memory Pool Increment Size

SYNOPSIS

```
extern int _mstep;
```

DESCRIPTION

This external integer is used by the memory allocation functions. It specifies the minimum amount of memory that will be allocated from the system for the local memory pool.

While additional memory is added to the local pool, it will not be contiguous with the memory already in the pool. If the additional amount is small, it can lead to severe fragmentation of the local pool. The memory allocation functions attempt to avoid this by rounding the amount needed up to the next multiple of the figure in **_mstep**.

This technique works well for small allocation requests. However, if your application requires mostly large blocks of memory, **_mstep** should be set to a small non-zero figure to allow for a more efficient allocation.

SEE ALSO

getmem, rlsmem

NAME

_OSERR DOS Error Information

SYNOPSIS

os_nerr Number of AmigaDOS error codes

os_errlist AmigaDOS error messages

```
#include <dos.h>
extern int _OSERR;
extern int os_nerr;
extern struct DOS_ERRS os_errlist[ ];
```

DESCRIPTION

The external integer named **_OSERR** contains error information returned by AmigaDOS after a system call has failed. In general, the Lattice library resets **_OSERR** at the beginning of any function that makes AmigaDOS system calls. Then if a system call fails during that function, the system error code is saved in **_OSERR**.

The AmigaDOS error number is mapped into an equivalent UNIX error number, which is placed in **errno**. If there is no appropriate UNIX number, **errno** will contain -1, defined symbolically as **EOSERR**. The function returns with a suitable error indication, which is usually -1 for functions that return integer values or NULL for functions that return pointers.

The **os_nerr** and **os_errlist** items are defined in a C source file named **oserr.c** and are used by the **poserr** function to print messages that correspond to the code found in **_OSERR**.

The following list applies to AmigaDOS 1.1 and is what we provide in **oserr.c**:

CODE	MEANING
103	Insufficient free store
104	Task table full
120	Argument line invalid or too long
121	File is not an object module
122	Invalid resident library during load
202	Object in use
203	Object already exists
204	Directory not found
205	Object not found
206	Invalid window
209	Packet request type unknown
210	Invalid stream component name
211	Invalid object lock
212	Object not of required type
213	Disk not validated
214	Disk write protected
215	Rename across devices attempted
216	Directory not empty
218	Device not mounted
219	Seek error
220	Comment too big
221	Disk full
222	File is protected from deletion
223	File is protected from writing
224	File is protected from reading
225	Not a DOS disk
226	No disk in drive
232	No more entries in directory

SEE ALSO

AmigaDOS Technical Reference Manual, poserr

_SLASH

Directory separator character *Class: LATTICE*

NAME

_SLASH Directory separator character

SYNOPSIS

```
extern char _SLASH;
```

DESCRIPTION

This external character is used by various functions which construct file names. It specifies the character to be used for separating components of the directory path. For AmigaDOS, the default is a forward slash (/), while for MSDOS it is a backslash (\).

SEE ALSO

strmfn, strmfp

Appendix E. Environment Variables

Appendix E describes the AmigaDOS environment variables that are used by the Latttice C Compiler and related programs.

include:

#include search path *Class: AMIGA*

NAME

include: #include search path

SYNOPSIS

```
assign INCLUDE: include path
```

DESCRIPTION

This logical name is used by the compiler to find files that are included via the **#include** statement. The **include path** should be the disk and directory specification that describe the location of the header files. For example,

```
assign INCLUDE: "Lattice C:include"
```

indicates that the compiler should look for included files in the directory **include** on the disk whose volume name is **Lattice C**. Note that if any portion of the path contains spaces, then the entire path must be enclosed in double quotes.

The compiler accepts two forms of the **#include** statement, as follows:

```
#include <file>

#include "file"
```

When the first form is used, the compiler looks for **file** only in the directory associated with the **INCLUDE:** logical name. With the second form, the compiler looks in the current directory, then in any directories specified via the **-i** option, and finally in the directory associated with the **INCLUDE:** logical name.

SEE ALSO

assign, lc, lc1

NAME

lc: Compiler executable file path

SYNOPSIS

```
assign LC: program path
```

DESCRIPTION

This is the logical name used by the driver program **lc** to locate the various programs required to compile and link source modules. This can be very useful on floppy disk based systems where there may not be sufficient online storage capacity to contain the compiler files in addition to other working files. By assigning **LC:** to the disk volume and directory containing the compiler and linker, **lc** will be able to locate and load the appropriate programs even if that disk volume isn't currently mounted. If required, you will be prompted to insert the necessary disk volume at the correct time.

For example,

```
assign LC: "Lattice C:"
```

will cause **lc** to load the compiler executable programs from the root level of the disk volume named **Lattice C**.

If the assignment is not made, **lc** looks for the programs in the default directory **lc/** in the root level of the system drive.

SEE ALSO

assign, lc

lib:

Library file path *Class: AMIGA*

SYNOPSIS

```
assign LIB: library path
```

DESCRIPTION

This is the logical name used by **lc** to locate the Lattice C startup routine **c.o** and the Lattice and Amiga libraries for the linker. If the assignment has been made, the driver program, **lc**, will specify that the linker look in the directory associated with the logical name for the startup code and the standard libraries. The directory may be on any disk volume. If the volume is not mounted, you will be prompted to insert the correct volume at the appropriate time. For example,

```
assign LIB: "Lattice C:lib"
```

specifies directory **lib/** on disk volume **Lattice C** as the location of the libraries and startup code.

If the assignment is not made, the linker will be directed to look in the subdirectory **lib/** of the path used to locate the linker. In other words, if the logical name **LC:** is assigned, the libraries will be assumed to be in the directory **LC:lib**. If **LC:** is not assigned, the directory **SYS:lc/lib** will be used.

SEE ALSO

assign, lc, lc1

NAME

quad: Intermediate file path

SYNOPSIS

```
assign QUAD: quadfile path
```

DESCRIPTION

This is the logical name used to direct the compiler to place the intermediate file in some location other than the source file directory. By default, the compiler creates the intermediate, or quad, file in the same directory as the source file. You can use the **-q** option of the **lc** driver program or the **-o** option of lc1 to cause this file to be created somewhere else.

The **QUAD:** logical name is used by the driver program to assign a different default location for the intermediate file. For example,

```
assign QUAD: ram:
```

will cause the compiler to use ram disk for the intermediate file when it is invoked via the **lc** command.

SEE ALSO

assign, lc, lc1

Appendix F. Function Names

Appendix F describes the public function names that are defined in the Lattice libraries.

abort

Abort the current process *Class: ANSI*

NAME

abort Abort the current process

SYNOPSIS

```
#include <stdlib.h>

abort();
```

DESCRIPTION

This function aborts the current process and returns a completion code of 3 to the parent process. Also the message *Abnormal program termination* is sent to **stderr**. Level 2 I/O buffers are not flushed.

RETURNS

The function does not return.

SEE ALSO

exit, _exit

NAME

abs	Absolute value
fabs	Float/double absolute value
iabs	Integer absolute value
labs	Long integer absolute value

SYNOPSIS

```
#include <math.h>
ax = abs(x);

ad = fabs(d);
ai = iabs(i);
al = labs(l);

double ad,d;
int ai,i;
long al,l;
```

DESCRIPTION

These functions compute the absolute value of the various numeric data types.

The most general approach is to use the **abs** macro, which is defined in the **math.h** header file. This macro accepts any data type as its argument and generates in-line code to perform the conversion. The definition is:

```
#define abs(x) ((x)<0?-(x):(x))
```

To minimize code size, you can use one of the function calls listed above instead of the **abs** macro. However, you must be careful to choose the function that

corresponds to the data type being converted. Note that **fabs** works with either a float or a double as its argument because float arguments are automatically promoted to double. Similarly, **iabs** will work with ints or shorts.

NAME

access Check file accessibility

SYNOPSIS

```
#include <stdio.h>

ret = access(name,mode);

int ret; return code
char *name; file name
int mode; access mode
```

DESCRIPTION

This function checks if a file is accessible in the way specified by **mode**, which follows the UNIX format:

0 = >	Check if file exists
1 = >	Check if file is executable
2 = >	Check if file is writable
3 = >	Check if file is writable and executable
4 = >	Check if file is readable
5 = >	Check if file is readable and executable
6 = >	Check if file is readable and writable
7 = >	Check if file is readable, writable, and executable

RETURNS

A return value of 0 indicates that access is allowed. If access is denied or the file cannot be found, -1 is returned. Additional error information can then be found in **errno** and **_OSERR**.

SEE ALSO

chmod, errno, _OSERR

NAME

argopt Get options from argument list

SYNOPSIS

```
#include <stdlib.h>

optd = argopt(argc,argv,opts,argn,optc);

char *optd; option data pointer
int argc; argument count
char *argv[]; argument vector
char *opts; options expecting data
int *argn; next argument number (changed)
char *optc; option character (changed)
```

DESCRIPTION

This function examines an argument list to find the next option argument, using the conventions similar to those of the UNIX *shell* command processor. These conventions are:

1. An option is an argument that begins with a slash (/) or a dash (i.e. a minus sign) and appears between the command verb (i.e. argv[0]) and the first non-option argument. The reason we recognize either a slash or a dash is that the former is a convention used by some AmigaDOS commands, while the latter has been used by UNIX for a long time.

2. The character immediately following the dash is called the *option character*, and it may be followed by a character string known as the *option data*.

3. If the option character appears in the **opts** string, then the data can be separated from the character by white space. In effect, this means that the data might be in the next **argv** entry if it does not follow the option character in the current entry.

4. A dash or slash followed by a blank or a dash indicates the end of the options.

Each time argopt is called, it will find the next option in the argument array and update the integer referenced by **argn**. On the first call, you should set this integer to 1, since **argv[0]** points to the command verb. The **argc** and **argv** items are normally the same as those passed to your main program, and they are not changed as a result of the **argopt** calls. The option character is returned in the byte referenced by **optc**, and the function returns a pointer to the option data string or to a null byte. If the next entry in **argv** is not an option, then the function returns a NULL pointer.

The **opts** item provides some flexibility in the way the option data is handled. If **opts** points to an empty string, then any option data must immediately follow the option character. However, if **opts** is not empty, then it lists the option characters that always have data. For those characters, the data can be preceded by white space in the command line. What this actually means is that **argopt** will look at the next entry in **argv** if the option character is not followed by a data string. If that next entry does not begin with a dash, then it is taken as the option data. See the examples below for clarification.

RETURNS

If the next argument is not an option, the function returns a NULL pointer. Otherwise, it returns a pointer to the option data, which will be an empty string if there was no data. Also, if an option was found, the character is placed into the byte referenced by **optc**, and **argn** is adjusted to index the next entry in **argv**.

SEE ALSO

main

EXAMPLES

```
/*
*
* Assume that this program is invoked by the following
command line:
*
* myprog -x -ypdq -z -g moo -g - blah
*
* The output will then be:
*
* Option: x Data:
* Option: y Data: pdq
* Option: z Data:
* Option: g Data: moo
* Option: g Data:
* Arg[8]: blah
*
*/
#include <stdlib.h>
char opts[ ] = "gx";
main(argc,argv)
int argc;
char *argv[ ];
{
char option,*odata;
int next;

for(next = 1; (odata = argopt(argc,argv,opts,&next,&option))
!= NULL); )
  printf("Option: %c, Data: %s\n",option,odata);

for(; next < argc; next++)
        printf("Arg[%d]: %s\n",next,argv[next]);
}
```

NAME

asctime Generate ASCII time string

SYNOPSIS

```
#include <time.h>

s = asctime(t);

char *s; points to time string
struct tm *t; points to time structure
```

DESCRIPTION

This function converts a time structure into an ASCII string of exactly 26 characters having the form:

```
DDD MMM dd hh:mm:ss YYYY\n\0
```

where **DDD** is the day of the week, **MMM** is the month, **dd** is the day of the month, **hh:mm:ss** is the hour:minute:seconds, and **YYYY** is the year. An example is:

```
Wed Sep 04 15:13:22 1985\n\0
```

The time pointer returned by the function refers to a static data area that is shared by both **ctime** or **asctime**. The time structure argument **t** is usually returned by the **gmtime** or **localtime** function.

SEE ALSO

gmtime, localtime

EXAMPLES

```
#include <time.h>
#include <stdio.h>

main()
{
struct tm *tp;
long t;

time(&t);
tp = localtime(&t);
printf("Current time is %s\n",asctime(tp));
}
```

NAME

assert	Assert program validity
_assert	Failure exit for assert

SYNOPSIS

```
#include <assert.h>

assert(x);
_assert(exp,file,line);

char *exp; failing expression
char *file; source file name
char *line; source line number
```

DESCRIPTION

The **assert** macro tests an expression **x** for validity (non-zero value). If the expression is 0, then the macro calls the **_assert** function with the expression in text form plus the source file name and line number, also as text strings. The default version of **_assert** prints a message on **stderr** and aborts with an exit code of 1. You can replace this function with one of your own if you require some other action when an assertion fails. The source code is supplied in the compiler package.

Note that the **assert.h** header file must be included in your program in order to define the macro. Also, the file contains two versions of the macro. If the symbol **NDEBUG** is defined, then a null version of the macro is used; otherwise the normal code-generating version applies. This allows you to strip the assertion code from your program without removing the **assert** calls. To do this, define **NDEBUG** in one of your header files or on the compiler command line via the **-d** option. In the former case, the header file containing the **NDEBUG** definition must be included before **assert.h**.

EXAMPLES

```
/* Make sure integer x is positive */

postest(x)
int x;
{
assert(x >= 0);
}
```

NAME

atof Convert ASCII to float

SYNOPSIS

```
#include <math.h>

d = atof(p);

double d; floating point result
char *p; input string pointer
```

DESCRIPTION

This function converts an ASCII input string into a double value. The string can contain leading white space and a plus or minus sign, followed by a valid floating point number in normal or scientific notation. If scientific notation is used, there can be no white space between the number and the exponent. For example,

```
123.456e-53
```

is a valid number in scientific notation.

EXAMPLES

```
/*
*
* This program tests the atof function.
*
*/
#include <stdio.h>
#include <math.h>

main()
{
char buff[80];
double d;
```

```
        while(1)
                {
                printf("\nEnter a number: ");
                if(gets(buff) == NULL) exit(0);
                if(buff[0] == '\0') exit(0);
                d = atof(buff);
                printf("%e\n",d);
                }
        }
```

NAME

atoi, atoil	Convert ASCII to integer
atoi	Convert ASCII to integer
atol	Convert ASCII to long integer

SYNOPSIS

```
#include <stdlib.h>

x = atoi(s);
y = atol(s);

int x;
long y;
char *s;
```

DESCRIPTION

These functions convert ASCII strings into normal or long integers. The string must have the form:

```
[whitespace][sign]digits
```

where **[whitespace]** indicates optional leading white space, **[sign]** indicates an optional + or - sign character, and **digits** is a contiguous string of digit characters. Once the digit portion is reached, the conversion continues until a non-digit character is hit. No check is made for integer overflow.

RETURNS

As noted above.

SEE ALSO

atof, stcd_i, stcd_l

NAME

calloc, free, malloc, realloc	Level 3 memory allocation
calloc	Allocate and clear Level 3 memory
free	Free Level 3 memory
malloc	Allocate Level 3 memory
realloc	Re-allocate Level 3 memory

SYNOPSIS

```
#include <stdlib.h>

b = calloc(nelt,esize);
b = malloc(n);
nb = realloc(b,n);
error = free(b);

char *b; block pointer
unsigned nelt; number of elements
unsigned esize; element size
unsigned n; number of bytes
char *nb; new block pointer
int error; 0 for success, -1 for failure
```

DESCRIPTION

These functions form Level 3 of Lattice's layered memory allocation system. This level is fully compatible with UNIX and with the ANSI standard.

The **malloc** function allocates a block that is **n** bytes long and is aligned in such a way

that you can cast the block pointer to any pointer type. If the block cannot be allocated, a NULL pointer is returned. The **calloc** function uses **malloc** to get a block whose size in bytes is given by

```
n = nelts * esize;
```

Then the block is cleared to zeroes. In common with **malloc ,calloc returns a NULL pointer if the block cannot be allocated.**

A call to **realloc** will obtain a new block whose size is **n** bytes. Then it copies the old block **b** to the new block **nb** and releases the old block. If **n** is greater than the old block size, the excess space is cleared to zeroes. If it is less than the old block size, only the first **n** bytes are copied. As described below, you can, under certain circumstances, re-allocate a block that has been freed.

The **free** function releases a block that was previously obtained via **calloc, malloc** or **realloc**. For compatibility with some versions of UNIX, the block is not actually returned to the free space pool until the next time you call **calloc, malloc, realloc** or **free**. Then if that next call is to **realloc** and the block being re-allocated is the one that was just freed, **realloc** will proceed correctly. In other words, you can ask **realloc** to re-allocate a block that was freed as long as you have not called **calloc, malloc** or **realloc** in the meantime.

RETURNS

For **calloc, malloc** and **realloc**, a NULL pointer is returned if there is not enough space for the requested block. For **free**, 0 is returned if the block was successfully released; otherwise, the function returns -1.

> NOTE: These functions are called *Level 3 memory allocation* because they call upon Level 2 functions **getmem** and **rlsmem**. You should be aware that the Level 1 function **rbrk** frees all Level 2 and Level 3 blocks. Be careful!

SEE ALSO

getmem, rlsmem, sbrk, rbrk

EXAMPLES

```
/*
*
* This program builds linked lists of text strings obtained
from the
* standard input file.
*
*/
#include <stdio.h>
#include <stdlib.h>
/*
*
* These elements are linked together to form the text string
list.
*
*/
struct LIST
  {
  struct LIST *next; /* forward linkage */
  char text[2];  /* minimum text string */
  };
/*
*
* Main program
*
*/
main()
{
struct LIST *p,*q,*list = NULL;
char b[256];
int x;
/*
* Build the list
*/
while(1)
  {
  printf("\nBegin new group...\n");
  for(q = (struct LIST *)(&list);;q = p)
    {
    printf("Enter a text string: ");
    if(gets(b) == NULL) break;
    if(b[0] == '\0') break;
```

```
        x = sizeof(struct LIST) - 2 + strlen(b) + 1;
        p = (struct LIST *)malloc(x);
        if(p == NULL)
          {
          printf("No more memory\n");
          break;
          }
        p->next = q->next;
        strcpy(p->text,b);
        }
    /*
    * Print the list
    */
      printf("\n\nTEXT LIST...\n");
      for(p = list; p != NULL; p = p->next)
        {
        printf("%s",p->text);
        free(p);
        }
    }
}
```

ceil, floor

Get floating point limits *Class: ANSI*

NAME

ceil, floor	Get floating point limits
ceil	Get ceiling of a real number
floor	Get floor of a real number

SYNOPSIS

```
#include <math.h>

x = ceil(y);
x = floor(y);

double x,y;
```

DESCRIPTION

These functions return the integral values that are nearest to the specified real number. For **ceil**, the return is the next higher integer, while **floor** returns the next lower integer.

> NOTE: Even though these functions return integral values, the results are still real numbers.

EXAMPLES

```
#include <math.h>

double r;

r = ceil(523.96); /* r contains 524.0 */
r = floor(523.96); /* r contains 523.0 */
```

NAME

chdir Change current directory

SYNOPSIS

```
#include <stdio.h>

error = chdir(path);
int error; 0 if successful
char *path; points to new directory path string
```

DESCRIPTION

This function changes the current directory to the specified path. For AmigaDOS, the path may begin with a drive or volume name and a colon.

RETURNS

If the return value is non-zero, then the operation failed. An AmigaDOS error code will be in **_OSERR**, and a UNIX error code will be in **errno**.

SEE ALSO

mkdir, rmdir, getcd, getcwd, errno, _OSERR

NAME

chkabort Check for Break character

SYNOPSIS

```
void chkabort();
```

DESCRIPTION

This function forces immediate checking for the break characters CTRL-C and CTRL-D. Normally this check only occurs when level 1 I/O is performed. The **chkabort** function provides a mechanism for detecting the break characters at other times. This can be important in programs which do little or no I/O processing.

If either break key is detected, the break trap is executed. The break trap is simply a function that gets called whenever the user keys CTRL-C or CTRL-D. If the break trap returns a zero value, control returns to the calling function. Otherwise the program terminates. The default handler displays a requester allowing the user to abort the program. It may be replaced via calls to **onbreak** or **signal**.

RETURNS

There is no return value.

SEE ALSO

signal, onbreak

NAME

chkml Check for largest memory block

SYNOPSIS

```
#include <stdlib.h>

size = chkml();
long size;
```

DESCRIPTION

This function returns the size, in bytes, of the largest block that is currently available in the level 2 memory pool without calling upon the operating system to supply additional heap space.

> NOTE: The return value is a long integer, so the function must be declared appropriately.

SEE ALSO

getmem, getml, rlsmem, rlsml, sizmem

NAME

chkufb Check Level 1 file handle

SYNOPSIS

```
#include <ios1.h>

ufb = chkufb(fh);

struct UFB *ufb; pointer to UNIX file block
int fh; file handle
```

DESCRIPTION

This function checks if an AmigaDOS file handle is currently associated with a Level 1 file. Normally it is used internally by **open**, **close**, **read**, **write**, **lseek** and **tell**. The UFB structure is defined in header file **ios1.h**. For AmigaDOS this structure is two integers. The first contains the mode flags specified in the call to the **open** function. The second integer contains the file handle. The external name **_ufbs** refers to an array of UFB structures, and the external integer **_nufbs** indicates how many structures are in the array. Normally this value is twenty.

RETURNS

If no UFB is currently attached to the file handle, a null pointer is returned.

NAME

chmod Change file protection mode

SYNOPSIS

```
#include <stdio.h>

error = chmod(name,mode);

int error; error code
char *name; file name
int mode; protection mode
```

DESCRIPTION

This function changes a file's protection mode. It is compatible with UNIX, although AmigaDOS also provides separate delete protection for each file. The **mode** argument should be formed by ORing any combination of the following symbols which are defined in **fcntl.h:**

Value	Meaning
S_IWRITE	Write permission
S_IREAD	Read permission
S_IEXECUTE	Execute permission
S_IDELETE	Delete permission

RETURNS

If the operation is successful, the function returns 0. Otherwise it returns -1 and places error information in **errno** and **_OSERR**.

SEE ALSO

access, errno, _OSERR

EXAMPLES

```
/*
*
* This example changes file "xyz/pdq.x" so it
* can be read and written.
*
*/
#include <stdlib.h>

if(chmod("xyz/pdq.x",S_IWRITE | S_IREAD))
  perror("Change mode");
```

NAME

clearerr, clrerr Clear Level 2 I/O error flag

clearerr Clear Level 2 I/O error flag

clrerr Clear Level 2 I/O error flag

SYNOPSIS

```
#include <stdio.h>

clrerr(fp);
clearerr(fp);

FILE *fp; file pointer
```

DESCRIPTION

These functions clear the error flag associated with the specified Level 2 file. Once set, the error flag forces an EOF return any time the file is accessed until the flag is reset.

Note that **clearerr** is a macro, while **clrerr** is a function. The former is provided for compatibility with some older versions of UNIX.

RETURNS

None.

Clear Level 2 I/O error flag *Class: ANSI*

SEE ALSO

fopen

NAME

close Close a Level 1 file

SYNOPSIS

```
#include <fcntl.h>

error = close(fh);

int error; non-zero if error
int fh; file handle
```

DESCRIPTION

This function closes a Level 1 file that was previously opened via the **open** function. Any pending output is completed and the file directory is updated.

All Level 1 files are automatically closed when your program terminates, but it is good programming practice to close a file when you are finished with it. One reason for doing this is to free up the operating system resources (e.g. control blocks and buffers) that are allocated for the file while it remains open.

RETURNS

The function returns 0 if it is successful. Otherwise, it returns -1 and places additional error information into **errno** and **_OSERR**.

SEE ALSO

errno, open, _OSERR

EXAMPLES

See the **open** function.

NAME

creat Create a Level 1 file

SYNOPSIS

```
#include <fnctl.h>

fh = creat(name,prot);

int fh; file handle
char *name; file name
int prot; protection mode
```

DESCRIPTION

This function is exactly the same as calling the **open** function in the following way:

```
open(name,O_WRONLY | O_TRUNC | O_CREAT , prot)
```

In other words, the file is created if it doesn't exist and truncated if it does exist. Then it is opened for writing. The protection mode can be any of the following:

Value	Meaning
S_IWRITE	Write permission
S_IREAD	Read permission
S_IWRITE | S_IREAD	Write and read permission
0	Write and read permission

In the current AmigaDOS implementation the protection mode is ignored.

RETURNS

If the operation is successful, the function returns a file handle, which is an integer equal to or greater than 0. Otherwise it returns -1 and places error information in **errno** and **_OSERR**.

SEE ALSO

errno, _OSERR, chgfa, chmod, close, open

NAME

ctime Convert time value to string

SYNOPSIS

```
#include <time.h>

s = ctime(t);
char *s; points to time string
long *t; points to time value
```

DESCRIPTION

This function converts a GMT time value to an ASCII string of exactly 26 characters having the form:

```
DDD MMM dd hh:mm:ss YYYY\n\0
```

where

DDD is the day of the week, **MMM** is the month, **dd** is the day of the month, **hh:mm:ss** is the hour:minute:seconds and **YYYY** is the year. An example is:

```
Wed Sep 04 15:13:22 1985n
```

The time pointer returned by the function refers to a static data area that is shared by both **ctime** or **asctime**.

The time value argument **t** must point to a long integer that is the number of seconds since 00:00:00 Greenwich Mean Time, January 1, 1970. Normally this value is obtained from the **time** function. Note that **ctime** converts this value back into local time by calling **tzset** and then subtracting the contents of **timezone**.

> NOTE: **t** is a pointer to a long integer. A common error is to pass the integer itself instead of the pointer. Observe the use of the ampersand (&) operator in the example above.

SEE ALSO

asctime, gmtime, localtime, time

EXAMPLES

```
#include <time.h>
#include <stdio.h>

main()
{
long t;

time(&t);
printf("Current time is %s\n",ctime(&t));
}
```

NAME

CXFERR Low-level float error

SYNOPSIS

```
#include <math.h>

CXFERR(code);
int code;
```

DESCRIPTION

This function is called when an error is detected by one of the low-level floating point routines, such as the arithmetic operations. Higher-level routines, such as the trigonometric functions, use the more sophisticated **matherr** function.

Programmers can replace this error trap with an application-dependent routine, as long as they still store the error code into the global integer **_FPERR**. This is necessary because some of the math functions check **_FPERR** to see if low-level errors occurred.

The error code passed to **CXFERR** indicates the type of floating point anomaly that occurred, as follows:

Symbol	Value	Meaning
FPEUND	1	Underflow
FPEOVF	2	Overflow
FPEZDV	3	Divide by zero
FPENAN	4	Not a number
FPECOM	5	Not comparable

These codes are defined in **math.h**.

CXFERR

Low-level float error *Class: LATTICE*

SEE ALSO

_FPERR, _FPA, matherr

NAME

dclose Close an AmigaDOS file

SYNOPSIS

```
#include <dos.h>

error = dclose(fh);

int error; 0 for success, -1 for error
int fh; file handle
```

DESCRIPTION

This function closes an AmigaDOS file that was opened via **dcreat, dcreatx** or **dopen.** AmigaDOS will not automatically close these files for you. One reason for doing this is to free up the operating system resources (e.g. control blocks and buffers) which are allocated for the file while it remains open.

If your program terminates normally, the run time support library will automatically close all files opened via level 1 or level 2 I/O calls.

RETURNS

If the operation is successful, the function returns 0. Otherwise it returns -1 and places error information in **errno** and **_OSERR.**

SEE ALSO

errno, _OSERR, dcreat, dcreatx, dopen, dread, dwrite, dseek

NAME

dcreat, dcreatx	Create AmigaDOS file
dcreat	Create or truncate AmigaDOS file
dcreatx	Create new AmigaDOS file

SYNOPSIS

```
#include <dos.h>

fh = dcreat(name,fatt);
fh = dcreatx(name,fatt);

int fh; file handle (-1 for error)
char *name; file name
int fatt; file attribute
```

DESCRIPTION

These functions create and open an AmigaDOS file, returning the file handle. The **dcreat** operation will truncate the file if it already exists, or create the file if it does not exist. Alternatively, **dcreatx** will fail if the file already exists.

The file attribute argument is accepted to provide source level compatibility with other systems, but is ignored under AmigaDOS.

RETURNS

If the operation is successful, the function returns a file handle. Otherwise it returns -1 and places error information in **errno** and **_OSERR**.

SEE ALSO

errno, _OSERR, dopen, dclose, dread, dwrite, dseek

dfind, dnext

Find directory entry *Class: AMIGA*

NAME

dfind, dnext	Find directory entry
dfind	Find first directory entry
dnext	Find next directory entry

SYNOPSIS

```
#include <dos.h>

error = dfind(info,name,attr);
error = dnext(info);

int error; 0 if successful
struct FILEINFO *info; file information area
char *name; file name or pattern
int attr; file attribute bits
```

DESCRIPTION

These functions search a directory for entries that match the specified file name or file name pattern. The **dfind** function locates the first matching file. Then successive calls to **dnext** locate additional matching files. Each **dnext** call must be given the file information that was returned on the preceding call to **dfind** or **dnext**.

The **name** argument must be a null-terminated string specifying the drive, path, and name of the desired file. The drive and path can be omitted, in which case the current directory will be searched. You can use the AmigaDOS wildcard characters as defined in the AmigaDOS Users Manual for pattern matching in the name portion. For example, *xy#?.b* will locate files in the current directory that begin with *xy* and have *b* as their extension. Setting the external integer location **msflag** to a nonzero value will cause all subsequent calls to **dfind** and **dnext** to use the MS-DOS * and ? characters for pattern matching in the name portion.

The **attr** argument specifies which file types are to be included in the search. If **attr** is zero, the search will include only normal files. Otherwise the search will also include directories.

The **info** argument points to a file information structure as defined in the **dos.h** header file. For AmigaDOS, this is the same as the AmigaDOS FileInfoBlock structure:

```
struct FileInfoBlock {
   LONG   fib_DiskKey;
   LONG   fib_DirEntryType;
   char   fib_FileName[108];
   LONG   fib_Protection;
   LONG   fib_EntryType;
   LONG   fib_Size;
   LONG   fib_NumBlocks;
   struct DateStamp fib_Date;
   char   fib_Comment[116];
};
```

RETURNS

If the operation is successful, a value of 0 is returned. Otherwise, the return value is -1, and further error information can be found in **errno** and **_OSERR**.

SEE ALSO

errno, _OSERR

dopen

Open an AmigaDOS file *Class: AMIGA*

NAME

dopen Open an AmigaDOS file

SYNOPSIS

```
#include <dos.h>

fh = dopen(name,mode);

int fh; file handle (-1 for error)
char *name; file name
int mode; access mode
```

DESCRIPTION

This function opens an AmigaDOS file and returns the file handle. The **mode** argument must be a mode supported directly by AmigaDOS and defined in the Amiga header file **libraries/dos.h**.

RETURNS

If the operation is successful, the function returns a file handle. Otherwise it returns -1 and places error information in **errno** and **_OSERR**.

SEE ALSO

errno, _OSERR, open, dcreat, dcreatx, dclose, dread, dwrite, dseek

NAME

drand	Generate random numbers
drand48	Random double (internal seed)
erand48	Random double (external seed)
lrand48	Random positive long (internal seed)
nrand48	Random positive long (external seed)
mrand48	Random long (internal seed)
jrand48	Random long (external seed)
srand48	Set high 32 bits of internal seed
seed48	Set all 48 bits of internal seed
lcong48	Set linear congruence parameters

SYNOPSIS

```
#include <math.h>

x = drand48();
x = erand48(seed);

y = lrand48();
y = nrand48(seed);

z = mrand48();
z = jrand48(seed);
```

```
srand48(hseed);
pseed = seed48(seed);
lcong48(parm);

double x; random double
long y; random positive long
long z; random long
short seed[3]; seed value (high bits in seed[0])
long hseed; high 32 bits of seed value
short *pseed; pointer to internal seed
short parm[7]; parameters
```

DESCRIPTION

These functions generate various types of random numbers using the linear congruential algorithm and 48-bit arithmetic. The normal functions **drand48**, **lrand48** and **mrand48** use an internal 48-bit storage area for the seed value. Special versions **erand48**, **jrand48** and **nrand48** are provided for cases where several seeds are in use at the same time, in which case the user specifies the seed on each function call.

The **drand48** and **erand48** functions return double values uniformly distributed over the interval from 0.0 up to but not including 1.0.

The **lrand48** and **nrand48** functions return non-negative long integers uniformly distributed over the interval from 0 to 2**31-1.

The **mrand48** and **jrand48** functions return signed long integers uniformly distributed over the interval from -2**31 to 2**31-1.

The **srand48** and **seed48** functions allow initialization of the internal 48-bit seed to something other than the default. For **srand48** the specified long value is copied into the high 32 bits of the seed, and the low 16 bits are set to 0x330E. For **seed48** the entire 48 bits are loaded from the specified array, and the function returns a pointer to the internal seed array.

The **lcong48** function allows a much more intricate initialization of the linear congruential algorithm. The algorithm is of the form:

```
X[n+1] = (a * X[n] + c) mod m
```

where **m** is 2**48 and the default values for **a** and **c** are 0x5DEECE66D and 0xB, respectively. The array passed to **lcong48** is structured as follows:

PARAMETER	VALUE
parm[0]	Bits 47-32 of value X[n]
parm[1]	Bits 31-16 of value X[n]
parm[2]	Bits 15-00 of value X[n]
parm[3]	Bits 47-32 of value a
parm[4]	Bits 31-16 of value a
parm[5]	Bits 15-00 of value a
parm[6]	value c

Whenever **seed48** is called, **a** and **c** are reset to their default values.

RETURNS

As noted above.

SEE ALSO

rand, srand

dread, dwrite

Read and write AmigaDOS files *Class: AMIGA*

NAME

dread, dwrite	Read and write AmigaDOS files
dread	Read from an AmigaDOS file
dwrite	Write to an AmigaDOS file

SYNOPSIS

```
#include <dos.h>

count = dread(fh,buffer,length);
count = dwrite(fh,buffer,length);

int count; actual bytes read or written
int fh; file handle
char *buffer; data buffer
int length; number of bytes to read or write
```

DESCRIPTION

These functions read or write an AmigaDOS file whose handle was returned by **dcreat**, **dcreatx** or **dopen**. Under normal circumstances, the value returned should match the buffer length. If this value is -1 or greater than the requested length, then some type of error occurred, and you should consult **errno** and **_OSERR**. If the actual length is less than the requested length when reading, this usually means that the file is exhausted. Similarly, if the actual length is less than the requested length for a write operation, this usually means that the device has no more space available. In both of these cases, it is still a good idea to check **errno** and **_OSERR** just in case some malfunction caused the short count.

RETURNS

If the operation is successful, the function returns the actual number of bytes transferred. Otherwise it returns -1 and places error information in **errno** and **_OSERR**.

SEE ALSO

errno, _OSERR, dcreat, dcreatx, dopen, dclose, dseek

NAME

dseek Re-position an AmigaDOS file

SYNOPSIS

```
#include <dos.h>

apos = dseek(fh,rpos,mode);

long apos; actual file position
int fh; file handle
long rpos; relative file position
int mode; seek mode
```

DESCRIPTION

This function re-positions an AmigaDOS file whose handle was returned by **dcreat**, **dcreatx** or **dopen**. The seek mode is the same as for **lseek** as follows:

Mode 0 Position is relative to beginning of file.

Mode 1 Position is relative to current file location.

Mode 2 Position is relative to end of file.

Note that for modes 1 and 2 **rpos** can be positive or negative, but **apos** is always the actual (positive) position relative to the beginning of file.

RETURNS

If the operation is successful, the function returns the actual file position, which is a long integer. Otherwise it returns -1L and places error information in **errno** and **_OSERR**.

SEE ALSO

errno, _OSERR, dcreat, dcreatx, dopen, dclose, dread, dwrite

NAME

ecvt, fcvt	Convert float to string
ecvt	Convert float to string
fcvt	Convert float to string

SYNOPSIS

```
#include <math.h>

s = ecvt(v,dig,decx,sign);
s = fcvt(v,dec,decx,sign);

char *s; string pointer
double v; floating point value
int dig; number of digits
int dec; number of decimal places
int *decx; pointer to decimal index (returned)
int *sign; pointer to sign indicator
```

DESCRIPTION

These functions convert a floating point number into an ASCII character string consisting of digits only and terminated by a null character.

For **ecvt**, the second argument indicates the total number of digits that should be generated, while for **fcvt** it indicates how many digits should be generated to the right of the decimal place. If the floating point value contains fewer significant digits, zeroes are appended. If there are too many significant digits, the low order (right-most) digit is rounded.

The **decx** argument points to an integer that will receive a value indicating where the decimal point should be placed in the string. For example, an index value of 3 indicates that the decimal point should be placed just after the third character in the string. A

value of zero means that the decimal point is just before the first character. If the index is negative, it indicates the number of zeroes that are between the decimal point and the first character. For example, -3 means that there are three zeroes between the decimal point and the beginning of the string.

The **sign** argument points to an integer that will be non-zero if v is negative.

EXAMPLES

```
#include <stdlib.h>

int decx,sign;
char *string;

string = ecvt(3.1415926535,10,&decx,&sign);

/*
*   string  =>  "3141592654"
*   decx    =>  1
*   sign    =>  0
*/

string = fcvt(3.1415926535,10,&decx,&sign);

/*
*   string  =>  "31415926535"
*   decx    =>  1
*   sign    =>  0
*/
```

NAME

exit, _exit	Terminate program execution
exit	Terminate with clean-up
_exit	Terminate with no clean-up

SYNOPSIS

```
#include <stdlib.h>

exit(code);
_exit(code);
```

DESCRIPTION

These functions terminate execution of the current program and return control to the parent program. Use **exit**, for a graceful termination, which means that all pending output buffers are written and all files are explicitly closed. The **_exit** function terminates immediately without writing output buffers or closing level 2 files. Generally, this latter form is used only in emergency situations when you do not care if some output data is lost.

In either case, files opened via **open**, **creat**, or **creatx** will be closed. However, files opened via **dopen**, **dcreat** or **dcreatx** will not be closed.

The parameter **code** is a value from 0 to 255 that gets passed back to the parent. By convention, a value of zero indicates success.

RETURNS

This function does not return.

EXAMPLES

```
/*
*
* This example shows how you would abort a program
* if it is not called with a valid input file name.
*
*/
#include <stdio.h>
main(argc,argv)
int argc;
char *argv[];
{
FILE *f;

if(argc > 1)
  {
  f = fopen(argv[1],"r");
  if(f == NULL)
    {
    fprintf(stderr,"Can't open file \"%s\"\n",argv[1]);
    exit(1);
    }
  }
else
  {
  fprintf(stderr,"No file specified\n");
  exit(1);
  }
```

NAME

exp	Exponential functions
exp	Exponential function
log	Natural logarithm function
log10	Base 10 logarithm function
pow	Power function
sqrt	Square root function

SYNOPSIS

```
#include <math.h>

r = exp(x); r = log(x); r = log10(x); r = pow(x,y); r = sqrt(x);

double r, x, y;
```

DESCRIPTION

The **exp** function raises the natural logarithm base E to the **x** power, and **pow** raises **x** to the **y** power. For **pow**, the **x** value must be an integer if it is negative. If it is not integral, **matherr** is called with a DOMAIN error.

The **log** and **log10** functions take the base E and base 10 logarithm, respectively. Each of these as well as **sqrt**, requires a positive argument. If a negative argument is supplied, **matherr** will be called with a DOMAIN error.

SEE ALSO

matherr

fclose, fcloseall

Close a Level 2 file *Class: ANSI*

NAME

fclose, fcloseall	Close a Level 2 file
fclose	Close a Level 2 file
fcloseall	Close all Level 2 files

SYNOPSIS

```
#include <stdio.h>

ret = fclose(fp);
num = fcloseall();

int ret; return code
int num; number of files closed
FILE *fp; file pointer for file to be closed
```

DESCRIPTION

The **fopen** function completes the processing of a Level 2 file and releases all related resources. If the file in the course of being written, any data which has accumulated in the buffer is written to the file, and the level 1 **close** function is called for the associated file descriptor. The buffer associated with the file block is freed. Even though **fclose** is automatically called for all open files when your program terminates or calls **exit**, it is considered professional programming practice to close your own files explicity. The last buffer is not written until **fclose** is called and data may be lost if an output file is not properly closed.

The **fcloseall** function closes all Level 2 files that were open and returns that number. However, if an error occurs on any file, **fcloseall** continues to close the other files and then returns a value of -1.

RETURNS

ret = -1	if error
ret = 0	if successful
num = -1	if error
num =	number of files closed if no errors

If -1 is returned, additional error information can be found in **errno** and **_OSERR**.

> NOTE: **fcloseall** closes the standard files **stdin**, **stdout** and **stderr**, This means, that functions such as **printf** and **perror** will fail after you call **fcloseall**.

SEE ALSO

fopen, errno, _OSERR

NAME

fdopen Attach Level 1 file to Level 2

SYNOPSIS

```
#include <stdio.h>

fp = fdopen(fh,mode);

FILE *fp; file pointer
int fh; file handle
char *mode; access mode
```

DESCRIPTION

This function attaches a level 1 file to level 2. In other words, if you have used **open** to prepare a file for level 1 I/O processing, you can subsequently use level 2 I/O with that file via **fdopen**. The file handle is the value returned by the **open** function, and the access mode has the same form as described for **fopen**.

RETURNS

If the operation is successful, the function returns a non-null file pointer. Alternatively, it returns a null pointer and places error information in **errno** and **_OSERR**.

SEE ALSO

fopen, errno, _OSERR

NAME

feof, ferror	Check Level 2 file status
feof	Check for Level 2 end-of-file
ferror	Check for Level 2 error

SYNOPSIS

```
#include <stdio.h>

ret = feof(fp);
ret = ferror(fp);

int ret; non-zero if condition is true
FILE *fp; file pointer
```

DESCRIPTION

These functions generate a non-zero value if the indicated condition is true for the specified file.

RETURNS

The return value is 0 if the condition is false, that is, no error for **ferror** or no end-of-file for **feof**. If the condition is true, a non-zero value is returned.

> NOTE: These functions are implemented as macros. Also, they do not check that **fp** is a valid file pointer.

fflush, flushall

Flush Level 2 output buffer *Class: ANSI*

NAME

fflush, flushall	Flush Level 2 output buffer
fflush	Flush a Level 2 output buffer
flushall	Flush all Level 2 output buffers

SYNOPSIS

```
#include <stdio.h>

ret = fflush(fp);
num = flushall();

FILE *fp; file pointer
int ret; return code
int num; number of open files
```

DESCRIPTION

The **fflush** macro flushes the output buffer of the specified Level 2 file. That is, it writes the buffer if the file is opened for output and the buffer contains any pending data. If an error occurs, the return value is EOF and the appropriate error code is placed into **errno**.

The **flushall** function flushes all Level 2 output buffers and returns the number of Level 2 files that are open. If an error occurs, the function continues to flush the remaining files and then returns a value of -1.

RETURNS

ret = 0 if successful
ret = EOF if error

num = number of open files if no error
num = -1 if error

In the event of a -1 return, error information can be found in **errno** and **_OSERR**.

SEE ALSO

fopen, fclose, errno, _OSERR

NAME

fgetc, getc, getchar	Get a character
fgetc	Get a character from a file
getc	Get a character from a file
getchar	Get a character from stdin
fgetchar	Get a character from stdin

SYNOPSIS

```
#include <stdio.h>

c = fgetc(fp);
c = getc(fp);
c = getchar();
c = fgetchar();

int c; return character or code
FILE *fp; file pointer
```

DESCRIPTION

These functions get a single character from the specified Level 2 file (**stdin** for **fgetchar** and **getchar**). Note that **getc** and **getchar** are actually implemented as macros in order to maximize execution speed.

Class: ANSI *Get a character*

RETURNS

r = next input character if successful
r = EOF if error or end of file

In the event of an EOF return, error information can be found in **errno** and **_OSERR**. Most programmers treat any EOF return as an indication of end-of-file. However, if you want to distinguish errors from an end-of-file condition, you should reset **errno** before calling the function and then analyze its contents when you receive an EOF return.

SEE ALSO

fopen, errno, _OSERR

NAME

fgets, gets	Get string from a Level 2 file
fgets	Get string from Level 2 file
gets	Get string from stdin

SYNOPSIS

```
#include <stdio.h>

p = fgets(buffer,length,fp);
p = gets(buffer);

char *p; buffer pointer or NULL
char *buffer; buffer pointer
int length; buffer length in bytes
FILE *fp; file pointer
```

DESCRIPTION

The **fgets** function gets a string from the specified Level 2 file, which must have been previously opened for input. Characters are copied from the file to the buffer until a newline ('\n') has been copied, or the buffer is full, or the end-of-file is hit. In the newline case, a null byte ('\0') is placed into the buffer after the newline if the buffer has room. In the end-of-file case, a null byte is placed into the buffer after the last byte that was read. If the end-of-file is hit before any bytes are read, a NULL pointer is returned.

The **gets** function copies characters from **stdin** (the standard input file) until a newline is reached. The newline is not copied to the buffer, but a null byte ('\0') is put there in its place.

NOTE: **fgets** will not return a null-terminated string if **length** characters have already been placed into the buffer. Also, make sure that your **gets** buffer can hold the largest line that will be encountered while reading **stdin**, because the function does not have any way to check for a maximum length.

RETURNS

Both functions return the **buffer** argument unless an end-of-file or I/O error occurs, in which case a NULL pointer is returned.

SEE ALSO

errno, fopen, feof, ferror, fgetc, getc

EXAMPLES

```
/*
*
* Assume that stdin contains the following lines:
*
* Hello, folks!
* Goodbye, folks!
*
*/
#include <stdio.h>

char *p,b[80];

/* For the next two lines, p will point to b */

p = gets(b);

/*  now b contains "Hello, folks!" */

p = fgets(b,sizeof(b),stdio);

/*  now b contains "Goodbye, folks!\n" */

/* After the next line, p is NULL */
```

Get string from a Level 2 file *Class: ANSI*

```
p = gets(b);
```

NAME

fileno Get file number for a Level 2 file

SYNOPSIS

```
#include <stdio.h>

fh = fileno(fp);

int fh; file handle
FILE *fp; file pointer
```

DESCRIPTION

This function returns the file handle (i.e. the file number) associated with the specified file pointer. The file pointer must be one that was returned by **fopen**, **freopen** or **fdopen**.

RETURNS

As noted above.

> This function is implemented as a macro. Also, it does not check that **fp** is a valid file pointer.

NAME

fmod, modf	Float modulus operations
fmod	Compute floating point modulus
modf	Split floating point value

SYNOPSIS

```
#include <math.h>

x = fmod(y,z);
x = modf(y,p);

double x,y,z,*p;
```

DESCRIPTION

The **fmod** function returns **y** if **z** is 0. Otherwise, it returns a value that has the same sign as **y**, is less than **z**, and satisfies the relationship:

```
y = (i * z) + x
```

where **i** is an integer. This is, what the expression:

```
x = y % z;
```

would produce if the % operation were defined for floating point numbers.

The **modf** function separates the integral and fractional parts of **y** and returns them as two doubles. The function return value is the fractional part, and the integral part is placed in the double pointed to by **p**. Both parts have the same sign as **y**. Note that the fractional part is the number that would be obtained by calling **fmod** in the following way:

```
x = fmod(y,1.0);
```

> NOTE: Ensure that the second argument of **modf** is a pointer to a double. A common error is to use a pointer to an integer.

EXAMPLES

```
#include <math.h>

double r,ff,fi;

r = fmod(5.7,1.5); /* r contains 1.2 */

ff = modf(r,&fi); /* ff contains 0.2 */
                  /* fi contains 1.0 */
```

NAME

fmode Change mode of Level 2 file

SYNOPSIS

```
#include <stdio.h>

fmode(fp,mode);

FILE *fp; file pointer
int mode; 0 => mode A
1 => mode B
```

DESCRIPTION

This function is used to change the translation mode of a Level 2 file that has been opened via **fopen**, **freopen** or **fdopen**. In mode A, carriage returns are deleted on input, and a carriage return is inserted before each line feed on output. In mode B, all data is transferred with no changes. Mode A also detects the CTRL-Z character (0x1A) as a logical end of file mark.

> NOTE: The file pointer is not checked for validity.

SEE ALSO

fopen, freopen, fdopen

NAME

fopen Open a Level 2 file

SYNOPSIS

```
#include <stdio.h>

fp = fopen(name, mode);

FILE *fp; file pointer
char *name; file name
char *mode; access mode
```

DESCRIPTION

This function opens a file for buffered access. The **name** string can be any valid file name and may include a device code and directory path. The **mode** string indicates how the file is to be processed, as follows:

MODE	CREATE	TRUNC	READ	WRITE	APPEND	TRANSLATE
r	No	No	Yes	No	No	Default
w	Yes	Yes	No	Yes	No	Default
a	Yes	No	No	No	Yes	Default
r+	No	No	Yes	Yes	No	Default
w+	Yes	No	Yes	Yes	No	Default
a+	Yes	No	Yes	No	Yes	Default
ra	No	No	Yes	No	No	Mode A
wa	Yes	Yes	No	Yes	No	Mode A
aa	Yes	No	No	No	Yes	Mode A
ra+	No	No	Yes	Yes	No	Mode A
wa+	Yes	No	Yes	Yes	No	Mode A
aa+	Yes	No	Yes	No	Yes	Mode A
rb	No	No	Yes	No	No	Mode B

wb	Yes	Yes	No	Yes	No	Mode B
ab	Yes	No	No	No	Yes	Mode B
rb+	No	No	Yes	Yes	No	Mode B
wb+	Yes	No	Yes	Yes	No	Mode B
ab+	Yes	No	Yes	No	Yes	Mode B

CREATE	Yes	The file will be created if it does not already exist.
CREATE	No	The function will fail if the file does not already exist.
TRUNC	Yes	If the file exists, it will be truncated (i.e. marked as empty).
TRUNC	No	If the file exists, its current contents will not be disturbed.
READ	Yes	The file can be read via functions such as **fread** and **fgetc**. Also, **fseek** can be used to position the file before reading.
READ	No	The file cannot be read.
WRITE	Yes	The file can be written via functions such as **fwrite** and **fputc**. Also, **fseek** can be used to positon the file before writing.
WRITE	No	The file cannot be written, but see APPEND below.
APPEND	Yes	The file can be written, but it is automatically positioned to the current end-of-file before each write operation. This effectively prevents existing data from being changed.
APPEND	No	Automatic positioning to the end-of-file is not done before a write operation. Also, writes are not allowed unless WRITE is *'Yes'*.
TRANSLATE	Default	The external integer **_fmode** is used to set mode A or mode B as follows:

```
IF (_fmode & 0x8000) set mode B
ELSE set mode A
```

For AmigaDOS, **_fmode** is normally 0x8000.

TRANSLATE Mode A — On a read operation, each carriage return character ('\r') is deleted, and the CTRL-Z character is treated as a logical end-of-file mark. On a write operation, each line feed character ('\n') is expanded to a carriage return followed by a line feed.

TRANSLATE Mode B — The data is unchanged as it is read or written.

If the file is successfully opened, the function returns a pointer to a *'buffered I/O control block'*, which is defined in the header file **stdio.h**. Normally you will not need to access any information in the control block directly, but you should be very careful not to disturb the block accidentally. A common C programming error is to accidentally mutilate one of these control blocks, which can cause garbage to be written into a file.

RETURNS

A NULL pointer is returned if the file cannot be opened. Consult **errno** and **_OSERR** for detailed error information.

SEE ALSO

fclose, fdopen, fgetc, fgets, fputc, fputs, fread, freopen, fwrite

NAME

fork	Create a child process
forkl	Fork with arg list
forkv	Fork with arg vector
wait	Wait for child process to complete
waitm	Wait for multiple child processes

SYNOPSIS

```
#include <dos.h>

error = forkl(prog,arg0,arg1,...,argn,NULL,env,procid);
error = forkv(prog,argv,env,procid);

cc = wait(procid);
complist = waitm(proclist);

int error; error code
char *prog; program name
char *arg0; argument #0
char *arg1; argument #1
char *argn; argument #n
char *argv[ ]; argument vector
struct FORKENV *env; pointer to pseudo environment structure (may
be NULL)
struct ProcID *procid; pointer to process ID structure
struct ProcID **proclist; address of pointer to linked list of
process ID's
struct ProcID *complist; pointer to linked list of completed
process ID's
```

DESCRIPTION

These functions create a *child process* by loading a new program as a concurrent process. When the child process completes, the current program (i.e. the *parent process*) can obtain its completion code via the *wait()* or *waitm()* function. The parent and child are multiprogrammed; that is, the parent continues to execute until it calls either the *wait()* or *waitm()* function.

You can specify the arguments for the new program in two ways. For the *list method*, the function call includes a list of argument string pointers terminated by a NULL pointer. For the *vector method*, the function call includes a single pointer to an array of argument string pointers, with the array being terminated by a NULL pointer. Following UNIX conventions, the first argument (i.e. arg0 or argv[0]) should be the program name and is normally the same as prog. Under AmigaDOS, the arguments are all concatenated into a pseudo-command line, with a blank separating adjacent arguments and a carriage return character at the end. The maximum size of this line is 255 bytes.

The pseudo environment pointer, if specified, contains optional data pertaining to default files and process execution as follows:

```
struct FORKENV {
  long priority;   /* new process priority  */
  long stack;    /* stack size for new process  */
  long stdin;    /* stdin file handle for new process */
  long stdout;    /* stdout file handle for new process */
  long console;    /* console window for new process  */
struct MsgPort *msgport;  /* msg port to receive termination
*/
        };    /* message from child   */
```

The child process will be supplied with default values for any field left NULL. In addition, a NULL pointer may be passed for this parameter, which will cause default values to be used for all items. If the default values are used, the child process is created with a priority of 5, a stack size of 4000 bytes, the current stdin, stdout, and console for the parent process, and a new message port is created to receive the termination message.

The new process executes as a CLI type task. This means it will be expecting file handles for stdin and stdout to be present, and a console task handler to exist. If the

parent process is running as a Workbench process then it is possible for none of these to exist for the child process to inherit. If the parent process may be invoked from Workbench then extra effort should be made to ensure the presence of file handles the child may require.

The optional message port field is provided to allow the parent process to detect when a child process has terminated while awaiting other events, such as Intuition menu events. The termination message format is similar to an Intuition message.

```
struct TermMsg {        /* termination message from child */
 struct Message msg;
  long class;         /* class == 0  */
  short type;         /* message type == 0 */
  struct Process *process; /* process ID of sending task */
  long ret;          /* return value  */
       };
```

When a termination message is received the parent process must still call the *wait()* function to remove the message and unload the child process. If the message was removed from the port it must be replaced by calling *PutMsg()* before calling *wait()*.

The *forkl()* and *forkv()* functions load the program from the current path using the AmigaDOS search procedure. Under Workbench version 1.1 only the current directory and the C: directory are searched for the program. Under Workbench 1.2, directories or devices that have been added to the path are also searched. Alternatively, a path specification can be included as part of the program name.

If the specified program file cannot be found, a -1 return is made, and additional error information can be found in errno and _OSERR. Upon succesful creation of the child process, the fork() and forkl() functions fill in the process ID structure. The address of this structure must be passed as the last argument. The process ID structure is defined in dos.h and has the following format:

```
struct ProcID {  /* packet returned from fork()  */
  struct ProcID *nextID; /* link to next packet  */
  struct Process *process; /* process ID of child  */
  int UserPortFlag;
  struct MsgPort *parent; /* termination msg destination */
  struct MsgPort *child; /* child process' task msg port */
  long seglist;  /* child process' segment list */
       };
```

The nextID field may be used to link process ID structures into a simple linked list for use with the waitm() function. The process field is the address of the process's task structure, and is value used with ROM Kernel functions such as *Signal()* that require a task ID parameter. The remaining fields are used by the *wait()* functions.

The *wait()* or *waitm()* function must be called for each child process to ensure that the child process terminates cleanly. The *wait()* function takes as its single argument a pointer to the ProcID structure for the child task. It waits for a termination message from one particular process, replying to and discarding any other messages that arrive at the message port. The function returns the child process's completion code. This is the value that was passed to the *exit()* function when the child terminated.

If any child processes share a message port, however, then *waitm()* should be called to ensure that no termination messages are lost. The *waitm()* function requires the address of a pointer to the first ProcID structure in a linked list of child process ProcID structures. It waits for one or more termination messages, removing the ProcID structure from the original linked list and inserting it into a linked list of terminated process ProcID structures. The completion code is placed in the UserPortFlag field of the structure. The function then returns a pointer to the first structure in the list of terminated process structures. Since the original list is updated, it may be reused to wait for the remaining child processes.

> NOTE: The *wait()* or *waitm()* function MUST be called for each child process before terminating the parent process. Otherwise, the child process will never be unloaded, and there is an excellent possibility that the system will crash.

SEE ALSO

AmigaDOS functions LoadSeg and CreateProc, exit, wait

EXAMPLES

```
/**
*
* This program creates a child process and displays the
* return code
* The child program name and arguments are taken from
* the command line
*
*/
#include "dos.h"
#include "stdio.h"

struct ProcID child;

void main(argc,argv)
int argc;
char *argv[ ];
{
int ret;
if (argc < 2)
        {
        printf("no program specified\n");
        printf("usage: fork program [arg1] [arg2] ...
[argn]\n");
        exit(0);
        }
printf("parent: beginning fork of %s\n",argv[1]);
if (forkv(argv[1],&argv[1],NULL,&child) == -1)
        {
        printf("error forking child\n");
        exit(0);
        }
else  ret = wait(&child);
printf("parent: %s finished, ret = %d\n",argv[1],ret);
}

/*
* This example forks multiple child processes
*/

#include "dos.h"

char *child1_argv[ ] =  "task1", /* program name */
                        "argument1", /* 1st argument */
                        "argument2", /* etc., etc.    */
                        NULL;

char *child2_argv[ ] =  "task2", /* program name */
```

```
                                "argument1", /* 1st argument */
                                "argument2", /* etc., etc.    */
                                NULL;

char *child3_argv[ ] =          "task3", /* program name */
                                "argument1", /* 1st argument */
                                "argument2", /* etc., etc.    */
                                NULL;
void main( )
{
struct ProcID *children, *terminated, *task;
struct ProcID child1, child2, child3;
int taskno;

forkv(child1_argv[0],child1 argv,NULL,&child1);
forkv(child2_argv[0],child2 argv,NULL,&child2);
forkv(child3_argv[0],child3 argv,NULL,&child3);

child2->nextID = &child3;
child1->nextID = &child2;
children = &child1;

while(children)     /* wait until no more children */
    {
    terminated = waitm(&children) /* must pass ADDRESS of
pointer */
    for (task = terminated; task != NULL; task =
task->nextID)
        {
        if (task == &child1) taskno = 1;
        else if (task == &child2) taskno = 2;
        else if (task == &child3) taskno = 3;
        printf("task %d terminated, value = %d\n",
                taskno,task->UserPortFlag);
        }
    }
}
```

fprintf

Formatted print *Class: ANSI*

NAME

fprintf	Formatted print
fprintf	Formatted print to a file
printf	Formatted printf to stdout
sprintf	Formatted print to storage

SYNOPSIS

```
#include <stdio.h>

length = fprintf(fp,fmt,arg1,arg2,...);
length = printf(fmt,arg1,arg2,...);
length = sprintf(s,fmt,arg1,arg2,...);

int length; number of characters generated
char *fmt; format string
FILE *fp; file pointer
char *s; storage pointer
```

See below for arg1, arg2, and so on.

DESCRIPTION

These functions generate a stream of ASCII characters by analyzing the format string and performing various conversion operations on the remaining arguments. The **printf** form sends the output stream to the Level 2 file named **stdout**, which is usually the user's screen (i.e. the "console"). The **fprintf** form is similar to **printf**, but it sends the stream to the Level 2 file specified by **fp**. Finally, the **sprintf** form places the output characters into the storage area whose address is given by s. You must ensure that this area is large enough to hold the maximum number of characters that might be generated. Note that **sprintf** also generates a null byte to terminate the stored string.

The **fmt** argument points to a string consisting of ordinary characters and conversion specifications. The ordinary characters are simply copied to the output, but each conversion specification is replaced by the results of the conversion. These results come from operating sequentially upon the arguments that follow **fmt**. That is, the first conversion specification operates upon **arg1**, the second operates upon **arg2**, and so on. In some cases, as described below, a conversion specification may process more than one argument.

Each conversion specification must begin with a percent character (%). If you want to place an ordinary percent into the output stream, precede it with another percent in the **fmt** string. That is, %% will send a single percent character to the output stream. If the percent is not preceded by a percent, then it introduces a conversion specification, as follows:

```
%[flags][width][.precision][l]type
```

where the brackets [...] indicate optional fields, and the fields have the following definitions:

flags	Controls output justification and the printing of signs, blanks, decimal places, and hexadecimal prefixes.
width	Specifies the *field width*, which is the minimum number of characters to be generated for this format item.
precision	Specifies the *field precision*, which is the required precision of numeric conversions or the maximum number of characters to be copied from a string, depending on the **type** field.
l	The letter l indicates that the argument is of *large size*.
type	Specifies the type of argument conversion to be done.

FLAGS

If any flag characters are used, they must appear after the percent and can be any of the following:

Minus (-)	This causes the result to be left-adjusted within the field specified by **width** or within the default width.
Plus (+)	This flag is used in conjunction with the various numeric conversion types to cause a plus or minus sign to be placed before the result. If it is absent, the sign character is generated only for a negative number.
Blank	This flag is similar to the plus, but it causes a leading blank for a positive number and a minus sign for a negative number. If both the plus and the blank flags are present, the plus takes precedence.
Sharp (#)	This flag causes special formatting. With the 'o', 'x' and 'X' types, the sharp flag prefixes any non-zero output with 0, 0x, or 0X, respectively. With the 'f', 'e' and 'E' types, the sharp flag forces the result to contain a decimal point. With the g and G types, the sharp flag forces the result to contain a decimal point and also prevents the elimination of trailing zeroes.

WIDTH

The **width** is a non-negative number that specifies the minimum field width. If fewer characters are generated by the conversion operation, the result is padded on the left or right (depending on the minus flag described above). A blank is used as the padding character unless **width** begins with a zero. In that case, zero-padding is performed. Note that **width** specifies the minimum field width, and it will not cause lengthy output to be truncated. Use the **precision** specifier for that purpose.

If you don't want to specify the field width as a constant in the format string, you can code it as an asterisk (*), with or without a leading zero. The asterisk indicates that the width value is an integer in the argument list. See the examples for more information on this technique.

PRECISION

The meaning of the **precision** item depends on the field type, as follows:

Type c	The precision item is ignored.
Types d, o, u, x and X	The precision is the minimum number of digits to appear. If fewer digits are generated, leading zeroes are supplied.
Types e, E and f	The precision is the number of digits to appear after the decimal point. If fewer digits are generated, trailing zeroes are supplied.
Types g and G	The precision is the maximum number of significant digits.
Type s	The precision is the maximum number of characters to be copied from the string.

As with the width item, you can use an asterisk for the precision to indicate that the value should be picked up from the next argument.

CONVERSION TYPE

The conversion type can be any of the following:

c = > Single character conversion	The associated argument must be an integer. The single character in the rightmost byte of the integer is copied to the output.
d = > Decimal integer conversion	The associated argument must be an integer, and the result is a string of digit characters preceded by a sign. If the plus and blank flags are absent, the sign is produced only for a negative integer. If the *large size* modifier is present, the argument is taken as a long integer.
e = > Double conversion *-d.dde-ddd*	The associated argument must be a double, and the result has the form

```
-d.dde-ddd
```

where **d** is a single decimal digit, **dd** is one or more digits, and **ddd** is an exponent of exactly three digits. The first minus sign is omitted if the floating point number is positive, and the second minus sign is omitted if the exponent is positive. The plus and blank flags dictate whether there will be a sign character emitted if the number is positive. The *large size* modifier is ignored.

E = > Double conversion
-d.ddE-ddd

This is exactly the same as type **e** except that the result has the form

```
-d.ddE-ddd
```

f = > Double conversion
-dd.dd

The associated argument must be a double, and the result has the form

```
-dd.dd
```

where **dd** indicates one or more decimal digits. The minus sign is omitted if the number is positive, but a sign character will still be generated if the plus or blank flag is present. The number of digits before the decimal point depends on the magnitude of the number, and the number after the decimal point depends on the requested precision. If no precision is specified, the default is six decimal places. If the precision is specified as 0, or if there are no non-zero digits to the right of the decimal point, then the decimal point is omitted.

g = > Double conversion, general form

The associated argument must be a double, and the result is in the **e** or **f** format, depending on which gives the most compact result. The **e** format is used only when the exponent is less than -4 or greater than the specified or default precision. Trailing zeroes are eliminated, and the decimal point appears only if any non-zero digits follow it.

G = > Double conversion, general form	This is identical to the **g** format, except that the **E** type is used instead of **e**.
o = > Octal integer conversion	The associated argument is taken as an unsigned integer, and it is converted to a string of octal digits. If the *large size* modifier is present, the argument must be a long integer.
p = > Pointer conversion	The associated argument is taken as a data pointer, and it is converted to hexadecimal representation. Under AmigaDOS, the pointer is printed as 8 hexadecimal digits, with leading zeroes if necessary.
P = > Pointer conversion	This is the same as the **p** format, except that upper case letters are used as hexadecimal digits.
s = > String conversion	The associated argument must point to a null-terminated character string. The string is copied to the output, but the null byte is not copied.
u = > Unsigned decimal integer conversion	The associated argument is taken as an unsigned integer, and it is converted to a string of decimal digits. If the *large size* modifier is present, the argument must be a long integer.
x = > Hexadecimal integer conversion	The associated argument is take as an unsigned integer, and it is converted to a string of hexadecimal digits with lower case letters. If the *large size* modifier is present, the argument is taken as a long integer.
X = > Hexadecimal integer conversion	This is the same as the **x** format, except that upper case letters are used as hexadecimal digits.

RETURNS

Each function returns the number of output characters generated. For **sprintf**, this number does not include the terminating null byte.

EXAMPLES

```
/*
*
* This example prints a message indicated whether
* the function argument is positive or negative.
* In the second "printf", the width and precision
* are 15 and 8, respectively.
*
*/
#include <stdio.h>

pneg(value)
double value;
{
char *sign;

if(value < 0) sign = "negative";
else sign = "not negative";

printf("The number %E is %s.\n",value,sign);

printf("The number %*.*E is %s.\n",value,15,8,sign);
}
```

NAME

fputc, putc, putchar, fputchar	Put a character
fputc	Put a character to a level 2 file
putc	Put a character to a level 2 file
putchar	Put a character to stdout
fputchar	Put a character to stdout

SYNOPSIS

```
#include <stdio.h>

r = fputc(c,fp);
r = putc(c,fp);
r = putchar(c);
r = fputchar(c);

int r; EOF or c
int c; Character to be output
FILE *fp; Level 2 file pointer
```

DESCRIPTION

These functions put a single character to the specified Level 2 file (**stdout** for **fputchar** and **putchar**). Note that **putc** and **putchar** are actually implemented as macros in order to maximize execution speed.

RETURNS

r = c if successful
r = EOF if error

For disk files, an EOF return usually means that the disk is full. However, this type of return can also occur if the device is write-protected or if a write error occurs. In any case, additional error information can be found in **errno** and **_OSERR**.

SEE ALSO

fopen, errno, _OSERR

NAME

fputs, puts	Put string to Level 2 file
fputs	Put string to Level 2 file
puts	Put string to stdout

SYNOPSIS

```
#include <stdio.h>

error = fputs(s,fp);
error = puts(s);

int error; non-zero if error
char *s; string pointer
FILE *fp; file pointer
```

DESCRIPTION

The **fputs** function copies string **s** to a level 2 file that was previously opened for output. The string must be terminated by a null byte, which is not copied.

The **gets** function copies string **s** to **stdout** (the standard output file). The terminating null byte is not copied, but a newline is sent after the string.

RETURNS

If an error occurs, the return value is -1; otherwise, it is 0. Additional error information can be found in **errno** and **_OSERR**.

SEE ALSO

errno, ferror, fopen, fputc

EXAMPLES

```
/*
*
* This examples writes the following two lines to stdout:
*
* This is the first line
* This is the second line
*
*/
#include <stdio.h>

puts("This is the first line");
fputs("This is ",stdout);
puts("the second line");
```

NAME

fread, fwrite	Read and write blocks
fread	Read blocks from a Level 2 file
fwrite	Write blocks to a Level 2 file

SYNOPSIS

```
#include <stdio.h>

a = fread(b,bsize,n,fp);
a = fwrite(b,bsize,n,fp);

int a; actual number of blocks
char *b; pointer to first block
int bsize; size of block in bytes
int n; maximum number of blocks
FILE *fp; file pointer
```

DESCRIPTION

These functions perform Level 2 I/O operations to read and write blocks of data. Each block contains **bsize** bytes and up to **n** blocks are stored into contiguous memory locations beginning at location **b**.

For **fread**, blocks are read until **n** have been stored or until the end-of-file is hit. If the end-of-file is hit in the middle of a block, that partial block will be stored in the **b** array, but it will not be included in the function return value. In other words, the return value indicates the number of complete blocks that were read.

For **fwrite**, blocks are written until **n** have been sent or until the output device cannot accept any more. If the output device becomes full in the middle of a block, a partial block will be written, but it will not be included in the function return value. In other words, the return value indicates the number of complete blocks that were written.

RETURNS

The functions return the number of complete blocks that were processed.

SEE ALSO

fopen, fclose, ferror, feof, fgetc, fputc, fseek

NAME

freopen Reopen a Level 2 file

SYNOPSIS

```
#include <stdio.h>

fpr = freopen(name, mode, fp);
FILE *fpr; file pointer after re-opening
char *name; file name
char *mode; access mode
FILE *fp; current file pointer
```

DESCRIPTION

This function reopens a Level 2 file. That is, it attaches a new file to a previously used file pointer. The previous file is automatically closed before the file pointer is reused. The name and mode arguments are the same as those for **fopen**.

RETURNS

fpr = NULL if error
fpr = fp if successful

> NOTE: The return code should be checked for NULL; the same errors as defined for *fopen* may occur. Also, for complete portability, do not assume that **fpr** and **fp** are identical. Use **fpr** to access the reopened file, not **fp**.

SEE ALSO

fopen, fdopen

NAME

frexp, ldexp	Split or combine float value
frexp	Split fraction and exponent
ldexp	Load exponent

SYNOPSIS

```
#include <math.h>

f = frexp(v,xp);
v = ldexp(m,x);

double f; fraction
double v; value
int *xp; exponent pointer
int x; exponent
```

DESCRIPTION

The **frexp** function splits floating point value **v** into its fraction (mantissa) and exponent parts. The mantissa is returned as a double whose absolute value is greater than or equal to 0.5 and less than 1.0. The exponent is returned as an integer whose absolute value is less than 1024.

The **ldexp** function adds the integer **x** to the exponent in **f**, which is the same as computing

```
v = f * (2 ** x)
```

Note that if **f** and **x** are the results of **frexp**, then **ldexp** performs the reverse operation. Also, if the absolute value of the resulting exponent is greater than 1023, then **matherr** will be called with an overflow or underflow error indication.

SEE ALSO

fmod, matherr, modf

NAME

fscanf	Formatted input conversions
fscanf	Formatted input from a file
scanf	Formatted input from stdin
sscanf	Formatted input from a string

SYNOPSIS

```
#include <stdio.h>

n = fscanf(fp,fmt,arg1,arg2,...);
n = scanf(fmt,arg1,arg2,...);
n = sscanf(ss,fmt,arg1,arg2,...);

int n; number of input items matched, or EOF
FILE *fp; file pointer (fscanf only)
char *ss; input string (sscanf only)
char *fmt; format string
---- *argx; pointers to input data areas
```

DESCRIPTION

These functions perform formatted input conversions on text obtained from the standard input file, a specified Level 2 file, or a string. The input characters are read and checked against the format string, which may contain any of the following:

White space	Any number of spaces, horizontal tabs, or newline characters will cause input to be read up to the next character that is not white space.

Ordinary characters

Any character that is not white space and is not the percent sign (%) must match the next input character. Use a double percent (%%) in the format string to match a single percent in the input. If there is not an exact match, scanning stops, and the function returns.

Conversion specification

This is multi-character sequence that indicates how the next input characters are to be converted. The form is:

```
%*nlt
```

where the various fields are defined as follows:

% — A percent sign introduces a conversion specifier. If you want to match a percent sign in the input, indicate this by a double percent (%%) in the format string.

* — The asterisk is optional. If present, it means that the conversion should be performed, but the result should not be stored. There should be no value pointer in the argument list for a suppressed conversion.

n — This is an optional decimal number that specifies the maximum input field width. This is used only with the **s** format.

l — The letter l is optional. If present, it indicates that a long conversion should be performed.

t — The **t** stands for one of the following format characters: c, d, e, f, g, i, n, o, s, u and x. These are described below.

If the conversion is successful and assignment is not suppressed, the result is placed into the corresponding argument. The argument list must contain a pointer to an

appropriate data item for each conversion specification that does not suppress assignment.

The function returns the number of conversion values that were assigned. This can be less than the number expected if the input characters do not agree with the format string. If an end-of-input is reached before any values are assigned, the return value is EOF.

The format characters listed above specify how the input characters are to be converted. Leading white space is skipped in all cases except the **c** conversion.

c = > character

The corresponding argument must point to a character. The next input character is moved to that destination. No white space is skipped.

d = > decimal number

The corresponding argument must point to an integer or to a long integer. The latter applies if the **d** is preceded by an **l**. The input characters should be decimal digits, optionally preceded by a plus or minus sign.

e,f,g = > floating point

These three types are identical. The corresponding argument must point to a float or a double. The latter applies if the type letter is preceded by an **l**. The input characters must consist of the following sequence:

- Optional leading white space.
- An optional plus (+) or minus (-).
- A sequence of decimal digits.
- An optional decimal point followed by 0 or more decimal digits.
- An optional exponent, consisting of the letter **e** or **E** followed by an optional plus or minus sign followed by 1 or more decimal digits.

This general form is shown below, where [...] indicates an optional part:

```
[whitespace][sign]digits[.digits][exponent]
```

n = > character count	No input characters are read. The corresponding argument must point to an integer into which is written the number of input characters read so far.
o = > octal number	An octal number is expected, and the corresponding argument should point to an integer, or to a long integer if the **o** is preceded by an **l**.
s = > string	A character string is expected, and the corresponding argument should point to a character array large enough to hold the string and a terminating null byte. The input string is terminated by white space or the end-of-input. Also, if a maximum field width is specified, the output array size should be at least that width plus 1, because the reading of input characters will stop at the field width even if no white space has been hit.
u = > unsigned number	An unsigned decimal number is expected, and the corresponding argument should point to an unsigned integer, or to an unsigned long integer if the **u** is preceded by an **l**.
x = > hexadecimal number	A hexadecimal number is expected, and the corresponding argument should point to an integer, or to a long integer if the **x** is preceded by an **l**. The hexadecimal number can begin with the characters *0x* or *0X*, and case is not significant for the hexadecimal letters.

RETURNS

The function returns the number of assignments that were made. For example, a return value of 3 indicates that conversion results were assigned to **arg1**, **arg2** and **arg3**.

NOTE: All of the result arguments (i.e. arg1, arg2, and so on) must be pointers. Also, you should not supply a pointer for any conversion specification that uses the * to suppress assignment.

fseek, ftell, rewind

Set or get Level 2 file position *Class: ANSI*

NAME

fseek, ftell, rewind	Set or get Level 2 file position
fseek	Set Level 2 file position
ftell	Get Level 2 file position
rewind	Seek to beginning of Level 2 file

SYNOPSIS

```
#include <stdio.h>

error = fseek(fp,rpos,mode);
apos = ftell(fp);
error = rewind(fp);

int error; non-zero if error
FILE *fp; file pointer
long rpos; relative file position
int mode; seek mode
long apos; absolute file position
```

DESCRIPTION

The **fseek** function moves the byte cursor of a Level 2 file to a new position. The **mode** argument must be one of the following:

0 The **rpos** argument is the number of bytes from the beginning of the file. This value must be positive.

1 The **rpos** argument is the number of bytes relative to the current position. This value can be positive or negative.

2 The **rpos** argument is the number of bytes relative to the end of the file. This value must be negative or zero.

The **rewind** function resets the specified file to its first byte and is equivalent to the following **fseek** call:

```
error = fseek(fp,0L,0);
```

rewind is implemented as a macro which calls **fseek**.

The **ftell** function returns a long value that is the current byte position in the file, relative to the beginning. It is equivalent to the following call:

```
apos = lseek(fp->_file,0L,1);
```

and it is implemented as a true function, not as a macro.

RETURNS

For **fseek**, a value of -1 is returned if an error occurs, and for **ftell** an error is indicated by a return value of -1L. In either case, **errno** and **_OSERR** contain additional error information.

SEE ALSO

fopen, errno, _OSERR, lseek, tell

NAME

gcvt Convert float to string

SYNOPSIS

```
#include <math.h>

p = gcvt(v,dig,buffer);

char *p; points to buffer
double v; floating point value
int dig; number of significant digits
char *buffer; output buffer
```

DESCRIPTION

This function converts the specified floating point value into a null-terminated string in the output buffer. The string will be in either of two formats. First **gcvt** attempts to produce **dig** significant digits in the FORTRAN F format. If that fails, it produces **dig** significant digits in the FORTRAN E format. Trailing zeroes will be eliminated if necessary.

RETURNS

The function returns a pointer to the buffer.

> NOTE: Ensure that the specified buffer is large enough when using this function.

SEE ALSO

ecvt, fcvt

EXAMPLES

```
/*
*
* This example displays 314150.0
*
*/
#include <stdlib.h>

char s[100];

printf("%s\n",gcvt(-3.1415e5,7,s));
```

NAME

getcd Get current directory

SYNOPSIS

```
#include <dos.h>

error = getcd(drive,path);

int error; 0 if successful
int drive; drive code
char *path; points to 64-byte path area
```

DESCRIPTION

This function gets the current directory path for the specified disk drive. The drive specification is retained in this implementation for compatibility with other implementations. However, only a value of 0 (indicating the current drive) is supported under AmigaDOS. Any other value is considered an error.

Note that the path area must be large enough to contain the expected path. The returned string will contain the entire path, including the volume name of the device.

RETURNS

If the operation is successful, the function returns 0. Otherwise it returns -1 and places error information in **errno** and **_OSERR**.

SEE ALSO

getcwd, errno, _OSERR

NAME

getclk, chgclk	Get or change system clock
getclk	Get system clock
chgclk	Change system clock

SYNOPSIS

```
#include <dos.h>

getclk(clock); error = chgclk(clock);

int error; unsigned char *clock;
```

DESCRIPTION

The **getclk** function obtains the current setting of the system clock and places it into an 8-byte array as follows:

```
clock[0] => day of week (0 for Sunday)
clock[1] => year - 1980
clock[2] => month (1 to 12)
clock[3] => day (1 to 31)
clock[4] => hour (0 to 23)
clock[5] => minute (0 to 59)
clock[6] => second (0 to 59)
clock[7] => hundredths (0 to 99)
```

The **chgclk** function changes the setting of the system clock to the value specified in the array.

RETURNS

If the array is invalid or the operation failed, **chgclk** returns a non-zero value.

> NOTE: If your machine is equipped with a hardware clock, its state is not necessarily changed by a call to **chgclk**.

SEE ALSO

errno, _OSERR

NAME

getcwd Get current working directory

SYNOPSIS

```
#include <stdio.h>

p = getcwd(b,size);

char *p; Same as b is successful, else NULL
char *b; Points to path buffer
int size; Size of path buffer
```

DESCRIPTION

This function obtains the path name for the current working directory. If the buffer pointer **b** is not null, then the path string is placed there if it will fit, and the return pointer **p** is the same as **b**. If **b** is null, then **malloc** is used to obtain a buffer of **size** bytes to hold the path string. In this latter case, you should use the **free** function to release the buffer when you are finished with it.

Note that the **getcd** function is often more efficient under AmigaDOS since it does not use memory allocation.

RETURNS

If the operation is successful, the function returns a pointer to the buffer. Otherwise it returns a null pointer and places error information in **errno** and **_OSERR**. Also, a null pointer is returned if the path string will not fit in the buffer or if a buffer cannot be allocated. In either of those cases, **errno** is unchanged, and **_OSERR** is reset.

SEE ALSO

getcd, errno, _OSERR

NAME

getdfs Get free disk space

SYNOPSIS

```
#include <dos.h>

error = getdfs(drive,info);

int error; 0 if successful
char *drive; drive or volume name
(NULL => current drive)
struct DISKINFO *info; disk information
```

DESCRIPTION

This function obtains information about the specified disk drive, including the amount of free space available. If a null pointer is passed as the drive name, information is obtained about the current drive. The DISKINFO structure is defined in **dos.h**. For AmigaDOS, this is the same as the AmigaDOS InfoData structure:

```
struct InfoData {
   LONG   id_NumSoftErrors;
   LONG   id_UnitNumber;
   LONG   id_DiskState;
   LONG   id_NumBlocks;
   LONG   id_NumBlocksUsed;
   LONG   id_BytesPerBlock;
   LONG   id_DiskType;
   BPTR   id_VolumeNode;
   LONG   id_InUse;
};
```

RETURNS

A return value of 0 indicates success. If the drive code is invalid or no disk is mounted on that drive, then the return value is -1. Note that no additional information is provided in **errno** or **_OSERR**.

EXAMPLES

```
/* Compute number of bytes available on current drive: */

#include <dos.h>
struct DISKINFO info;
long size;

if(getdfs(0,&info) == 0)
    size = (info.id_NumBlocks - info.id_NumBlocksUsed)
             * info.id_BytesPerBlock;
```

NAME

getfa Get file attribute

SYNOPSIS

```
#include <dos.h>

fa = getfa(name);

int fa; 1, -1, or 0
char *name; file name
```

DESCRIPTION

This function determines whether or not the specified file is a directory. The status is returned in **fa** and contains the following information:

```
 1 => Directory file
 0 => Error
-1 => Normal file
```

RETURNS

If the operation is unsuccessful, the function returns 0 and places error information in **errno** and **_OSERR**.

SEE ALSO

errno, _OSERR

NAME

getfnl Get filename list

SYNOPSIS

```
#include <stdlib.h>

n = getfnl(fnp,fna,fnasize,attr);

int n; number of matching file names
char *fnp; file name pattern
char *fna; file name array
unsigned fnasize; size of file name array
int attr; file attribute
```

DESCRIPTION

This function gets all file names that match the specified pattern and attribute, and it places them into the file name array. Each name is stored as a null-terminated string, and the file name array is terminated by a null string (i.e. a string consisting of only a null byte). If the file name pattern includes a path prefix, that prefix is placed in front of each matching file name.

The function return value is the number of strings stored in the array, not including the terminating null string.

The file name pattern has the general form

```
drive:path/node
```

The function first strips off the drive and directory path portion and restricts its search to that area of the file system. The node and extension parts can contain any valid file name characters, including the wildcard pattern matching characters. AmigaDOS wildcard characters are supported by default. Setting the external integer location **msflag** to a nonzero value will cause MS-DOS wildcard characters * and ? to be supported instead. Some examples are:

a:#?.c	Finds all files on drive A that have *.c* as their extension. A file named *abc.c* would thus be place in the array as *a:abc.c*.
:abc/def/q.x?*	Finds all files in the directory **:abc/def** that begin with the letter q and have extensions consisting of the letter x and one other letter. For example, one such name would be */abc/def/queen.x*.
XYZ.*	Finds all files in the current directory that begin with *XYZ* and have no extension. One example is *XYZ*.

Notice that AmigaDOS makes no distinction between upper and lower case in any part of the file name.

The attribute is an integer defined as follows:

```
0 => non directory files
1 => all files including directories
```

RETURNS

A value of -1 is returned if the file name pattern is invalid or if there is not enough room in the file name array. In the first case, **_oserr** will contain further error information.

SEE ALSO

dfind, dnext, strbpl, strsrt, _oserr

EXAMPLES

```
/*
* This program constructs an array of pointers
*  to all normal files in the current directory
* that have an extension of .c.  Then
* the array is sorted into ASCII order.
*
*/
```

```
char names[3000],*pointers[300];
int count;

count = getfnl("#?.c",names,sizeof(names),0);

if(count > 0)
    {
    if(strbpl(pointers,300,names) != count)
        {
        fprintf(stderr,"Too many file names\n");
        exit(1);
        }
    strsrt(pointers,count);
    }
else
    {
    if(_OSERR) poserr("FILES");
    else fprintf(stderr,"Too many files\n");
    exit(1);
    }
```

NAME

getft Get file time

getft Get file time

SYNOPSIS

```
#include <dos.h>

ft = getft(name);

long ft; file time or -1 if error;
char *name; file name
```

DESCRIPTION

This function gets the time and date information associated with the specified file. This information usually indicates when the file was created or last updated. This function returns the file time expressed as the number of seconds since 00:00:00 Greenwich Mean Time, January 1, 1970.

RETURNS

If **getft** is successful, the file time (a long integer) is returned. Otherwise a value of -1L is returned. Additional error information can be found in **errno** and **_OSERR**.

SEE ALSO

errno, _OSERR

NAME

getmem, getml	Get Level 2 memory block
getmem	Get Level 2 memory block (short)
getml	Get Level 2 memory block (long)

SYNOPSIS

```
#include <stdlib.h>

p = getmem(sbytes);
p = getml(lbytes);

char *p; block pointer
unsigned sbytes; number of bytes
long lbytes; number of bytes
```

DESCRIPTION

These functions allocate a block from the Level 2 memory pool and return a pointer to the first byte in the block. If the pool does not currently contain a block of sufficient size, the Level 1 memory allocator, **sbrk**, is called to obtain more space from the operating system. If that step fails, a NULL pointer is returned.

Unlike the Level 3 memory allocator, Level 2 does not allocate any additional bytes to retain the block size and other overhead information. However, Level 2 does require a minimum block size of **MELTSIZE** bytes, where **MELTSIZE** is defined in the **dos.h** header file. If you request a block that is smaller, **getmem** and **getml** will round it up to the minimum size.

RETURNS

A NULL pointer is returned if the block could not be allocated. Otherwise, a character pointer is returned, but it can be cast to any other pointer type.

> NOTE: The Level 2 memory allocator does not retain any information about the blocks it has allocated, you must keep track of the block pointer and size if you plan to release the memory at some later time.

SEE ALSO

rlsmem, rlsml, sizmem

EXAMPLES

```
/*
*
* This program builds a linked list of text strings
* obtained from the standard input file.
*
*/
#include <stdio.h>
#include <stdlib.h>
/*
* These elements are linked together to form the
* text string list.
*/
struct LIST
  {
  struct LIST *next; /* forward linkage */
  int size;  /* element size */
  char text[2];  /* minimum text string */
  };
/*
*
* Main program
*
*/
main()
{
struct LIST *p,*q,*list = NULL;
```

```
char b[256];
int x;
/*
*
* Build the list
*
*/
for(q = (struct LIST *)(&list);;q = p)
  {
  printf("Enter a text string: ");
  if(gets(b) == NULL) break;
  if(b[0] == '
  x = sizeof(struct LIST) - 2 + strlen(b) + 1;
  p = (struct LIST *)getmem(x);
  if(p == NULL)
    {
    printf("No more memory\n");
    break;
    }
  p->next = q->next;
  p->size = x;
  strcpy(p->text,b);
  }
/*
*
* Print the list
*
*/
printf("\n\nTEXT LIST...\n");
for(p = list; p != NULL; p = p->next)
   printf("%s",p->text);
}
```

NAME

gmtime	Unpack Greenwich Mean Time (GMT)
localtime	Unpack local time

SYNOPSIS

```
#include <time.h>

ut = gmtime(t);
ut = localtime(t);

struct tm *ut;
long *t;
```

DESCRIPTION

These functions unpack a time value from the long integer form into a structure. Normally the time value represents the number of seconds since 00:00:00, January 1, 1970, Greenwich Mean Time. The **time** function returns this kind of number. For **gmtime**, this number is converted as is, while **localtime** adjusts the number for the local time zone.

SEE ALSO

asctime, ctime, localtime, time

> NOTE: These functions expect a pointer as the argument. A common error is to pass the actual time value instead of the pointer. Also, the functions share a static data area for their return values, and a call to either one will destroy the results of the previous call.

EXAMPLES

```
#include <time.h>
main()
{
struct tm *p;
long t;

time(&t);
p = gmtime(&t);
printf("GMT is %s\n",asctime(p));
}
```

NAME

iomode Change mode of Level 1 file

SYNOPSIS

```
#include <fcntl.h>

error = iomode(fh,mode);

int error; error code
int fh; file handle
int mode; 0 => translated mode
1 => raw mode
```

DESCRIPTION

This function changes the mode of the specified Level 1 file. When in translated mode, carriage returns are deleted on input, and a carriage return is inserted before each line feed on output. In raw mode, all data in the file is transferred as is.

Note that **iomode** affects only the software translation that is done by the Level 1 I/O service functions.

RETURNS

A non-zero return value indicates that the specified file handle is not associated with a Level 1 file.

SEE ALSO

open, getfc

NAME

is	Character tests
isalnum	Test if alphanumeric character
isalpha	Test if alphabetic character
isascii	Test if ASCII character
iscntrl	Test if control character
iscsym	Test if C symbol character
iscsymf	Test if C symbol lead character
isdigit	Test if decimal digit character
isgraph	Test if graphic character
islower	Test if lower case character
isprint	Test if printable character
ispunct	Test if punctuation character
isspace	Test if space character
isupper	Test if upper case character
isxdigit	Test if hex digit character

SYNOPSIS

```
#include <ctype.h>

t = isalnum(c);
t = isalpha(c);
t = isascii(c);
t = iscntrl(c);
t = iscsym(c);
t = iscsymf(c);
t = isdigit(c);
t = isgraph(c);
t = islower(c);
t = isprint(c);
t = ispunct(c);
t = isspace(c);
t = isupper(c);
t = isxdigit(c);

int t; truth value (0 if false, non-zero if true)
int c: character to test
```

DESCRIPTION

These functions test for various character types. If you include **ctype.h** as shown above, then the functions are actually defined as macros and generate in-line code to test the static array named **_ctype**. This array contains a bit mask for each of the 256 possible character values and for the integer value -1. See the **ctype.h** description for an explanation of this array.

If you don't include **ctype.h**, these functions will be resolved in the library, which can reduce your program size slightly at the expense of execution speed. If you want to use the function versions but must include **ctype.h** for some other reason, use **#undef** to undefine the appropriate character test macros.

NOTE: You can use either characters or integers as arguments, but the macros are defined only over the integer range from -1 to 255. The functions, however, will correctly handle the entire integer range.

The reason -1 is included as a valid argument is to avoid a nonsensical result if you feed the EOF value to one of the macros or functions. EOF can be returned by **getchar** and other I/O functions, and if you pass it to any of the character test functions, the resulting truth value will be zero.

SEE ALSO

_ctype

EXAMPLES

```
#include <stdio.h>
#include <ctype.h>
main()
char b[100];
int c;

while((c = getchar()) != EOF)
        printf("\n%c %s alphabetic.\n",c,
                isalph(c) ? "is" : "is not");
}
```

NAME

longjmp, setjmp	Perform long jump
longjmp	Perform long jump
setjmp	Set long jump parameters

SYNOPSIS

```
#include <setjmp.h>

ret = setjmp(save);
longjmp(save,value);

int ret; return code
int value; return value
jmp_buf *save; address of save area
```

DESCRIPTION

The **setjmp** function checkpoints the current stack mark in the save area and returns a code of 0. A subsequent call to **longjmp** will then cause control to return to the next statement after the original **setjmp** call, with **value** as the return code. If **value** is 0, it is forced to 1 by **longjmp**.

This mechanism is useful for quickly popping back up through multiple layers of function calls under exceptional circumstances.

RETURNS

A return code of 0 from **setjmp** indicates that this is the initial call to save the stack. A non-zero return code indicates that **longjmp** has been executed.

longjmp, setjmp

Perform long jump *Class: ANSI*

NOTE: Calling **longjmp** with an invalid save area is an effective way to disrupt the system. One common error is to use **longjmp** after the function calling **setjmp** has returned to its caller. This cannot possibly succeed, since the stack frame for that function no longer exists.

NAME

lsbrk, rbrk, sbrk	Level 1 memory allocation
lsbrk	Allocate Level 1 memory (long)
rbrk	Release Level 1 memory
sbrk	Allocate Level 1 memory (unsigned)

SYNOPSIS

```
#include <stdlib.h>

p = lsbrk(lbytes);
error = rbrk();
p = sbrk(sbytes);

char *p; block pointer
long lbytes; number of bytes
unsigned sbytes; number of bytes
int error; 0 if successful
```

DESCRIPTION

These functions form Level 1 of Lattice's layered memory allocation system. This level is **incompatible** with most versions of UNIX. The memory pool is viewed as a group of non-contiguous areas of memory allocated from the system free memory pool.

The **sbrk** and **lsbrk** functions requests **sbytes** or **lbytes** of memory from the system, adding the the allocated block to a linked list of memory blocks to be returned to the system when the program terminates. The function returns the address of the block just allocated.

The **rbrk** function returns all memory from this area to the operating system.

NOTE: Calling **rbrk** will free all memory in the memory pool, including level-2 I/O buffers.

RETURNS

If **sbrk** fails, it returns value **(char *)(-1)** which is -1 cast into a character pointer. This strange return is a legacy of UNIX. For **lsbrk**, an error is indicated by a NULL pointer.

SEE ALSO

getmem, malloc

Class: UNIX *Set or get file position*

NAME

lseek, tell	Set or get file position
lseek	Set Level 1 file position
tell	Get Level 1 file position

SYNOPSIS

```
#include <fcntl.h>

apos = lseek(fh,rpos,mode);
apos = tell(fh);

int fh; file handle
long rpos; relative file position
int mode; seek mode
long apos; absolute file position
```

DESCRIPTION

The **lseek** function moves the byte cursor of a Level 1 file to a new position. The **mode** argument must be one of the following:

0 The **rpos** argument is the number of bytes from the beginning of the file. This value must be positive.

1 The **rpos** argument is the number of bytes relative to the current position. This value can be positive or negative.

2 The **rpos** argument is the number of bytes relative to the end of the file. This value must be negative or zero.

If **lseek** is asked to move 0 bytes relative to the current position, it simply returns the current file position. The **tell** function is then equivalent to:

lseek, tell

Set or get file position *Class: UNIX*

```
apos = lseek(fh,0L,1);
```

RETURNS

Both functions return -1L if an error occurs, in which case **errno** and **_OSERR** contain additional error information.

SEE ALSO

MSDOS function 0x42, errno, _OSERR, open

EXAMPLES

```
/**
*
* This program totals the number of bytes used by
* all normal files in the current directory.
*
**/
#include <fcntl.h>  /* for Level 1 I/O */

char names[8192];  /* holds file names */

main()
{
char *p;
int f,n;
long x,y;

if(getfnl("*.*",names,sizeof(names),0) <= 0)
  {
  printf("Can't build file name list\n");
  exit(1);
  }
for(x = 0, n = 0, p = names; *p != '\0'; p += strlen(p) + 1)
  {
  f = open(p,O_RDONLY);
  if(f < 0)
    {
    printf("Can't open \"%s\"\n",p);
    exit(1);
    }
  y = lseek(f,0L,2);
```

```
    if(y < 0)
      {
      printf("Seek failure on \"%s\"\n",p);
      exit(1);
      }
    x += y;
    n++;
    close(f);
    }
  printf("%d files, %ld bytes used\n",n,x);
  }
```

NAME

main Your main or principle program

SYNOPSIS

```
#include "workbench/startup.h"

void main(argc,argv);

int argc; argument count
union {
char *args[ ];
struct WBStartup *msg;
} argv; argument vector
```

DESCRIPTION

This function does not actually exist in the library; you must supply one of these *main programs* in each of your applications. If you trace through the two startup modules *C.A* and *_MAIN.C*, you will find that *C.A* passes control to *_MAIN.C*, which then calls the function named **main**. Since we supply the source code for both of these modules, you are free to change this initialization procedure for special applications. The standard version simulates UNIX's interface with C programs by setting up a *vector*, which is simply an array of pointers.

The **argv.args** array contains pointers to the command line arguments, and **argc** indicates how many pointers are in the array. For example, if you invoke **myprog** with the following command line

```
myprog abc def "ghi jkl"
```

then **argv.args** is set up as follows:

```
argv.args[0] => "myprog"
argv.args[1] => "abc"
argv.args[2] => "def"
argv.args[3] => "ghi jkl"
```

and **argc** contains the value 4.

Under WorkBench there is no command line. In this case, **argc** is 0 indicating no command or arguments, and **argv.msg** is a pointer to the WorkBench startup message structure.

RETURNS

None. When **main** returns to its caller (normally *_MAIN.C*), the program exits to AmigaDOS with a termination code of 0. If you want to pass a non-zero termination code back to AmigaDOS, use the **exit** or **_exit** function.

SEE ALSO

exit, _exit

EXAMPLES

```
/**
*
* This program is intended to run only under CLI,
* and displays the command and any arguments
*
**/
void main(argc, argv)
int argc;
char *argv[ ];
{
int i;

printf("command = %s\n",argv[0]);
for (i = 0; argc > 1; i++, argc--)
        printf("argument %d = %s\n",i,argv[i]);
}

/**
*
* This program is intended to run only under WorkBench,
* and gets its arguments from the WorkBench message structure
*
**/
```

main

Your main or principle program *Class: ANSI*

```
#include "workbench/startup.h"

void main(argc, argv)
int argc;
struct WBStartup *argv;
{
int i;

if (argc != 0) exit(ERROR);

printf("command = %s\n",argv->sm_ArgList[0]->wa_name);
for (i = 1; i < argv->sm NumArgs; i++)
 printf("argument %d = %s\n",i,argv->sm_ArgList[i]->wa_name);
}

/**
*
* This program is intended to run under either
* WorkBench or CLI.
*
**/
#include "workbench/startup.h"

void main(argc, argv)
int argc;
union  {
        char *args[ ];
        struct WBStartup *msg;
        } argv;  argument vector
{
int i;

if (argc != 0)
  {
  printf("command = %s\n",argv.args[0]);
  for (i = 0; argc > 1; i++, argc--)
      printf("argument %d = %s\n",i,argv.args[i]);
  }
else
  {
  printf("command = %s\n",
          argv.msg->sm_ArgList[0]->wa_name);
  for (i = 1; i < argv.msg->sm_NumArgs; i++)
      printf("argument %d = %s\n",i,
              argv.msg->sm_ArgList[i]->wa_name);
   }
}
```

NAME

matherr Math error handler

except Call math error handler

SYNOPSIS

```
#include <math.h>

a = matherr(x);
r = except(type,name,arg1,arg2,retval);

int a; action code
struct exception *x; exception vector
double r; actual return value
int type; error type
char *name; math function name
double arg1; first argument
double arg2; second argument
double retval; proposed return value
```

DESCRIPTION

The **matherr** function is called whenever one of the higher-level math functions detects an error. The exception vector structure is defined in **math.h** and contains information about the error as follows:

```
struct exception
  {
  int type;  /* error type */
  char *name;  /* math function name */
  double arg1, arg2;  /* function arguments */
  double retval; /* proposed return value */
  };
```

The standard library version of **matherr** translates the error type into a UNIX error

code that is placed into **errno**. Then the function returns an action code of 0 to indicate that the math function should simply use the proposed return value. In other words, the math function will pass that value back to its caller.

The Lattice compiler package includes source code for **matherr** so you can change it to do more sophisticated error correction. One typical change is to place a different return value into the exception vector and then return a non-zero action code. This informs the math function that the return value has been changed.

The **except** function is a Lattice extension to UNIX that simplifies the interface to **matherr** by setting up the exception vector and processing the action code and return value. It is intended to ease the error-handling chore in user-written math functions.

When your math function encounters an error, it should call **except** specifying one of the following error types, which are defined in the **math.h** header file:

Symbol	Code	Meaning
DOMAIN	1	Domain error
SING	2	Singularity
OVERFLOW	3	Overflow (number too large)
UNDERFLOW	4	Underflow (number too small)
TLOSS	5	Total loss of significance
PLOSS	6	Partial loss of significance

You can define new type codes if your application requires them, but you should then change **matherr** to perform the appropriate mapping into the UNIX error codes. The default mapping is:

Matherr	Errno
DOMAIN	EDOM
SING	EDOM
OVERFLOW	ERANGE
UNDERFLOW	ERANGE
TLOSS	ERANGE
PLOSS	ERANGE

RETURNS

For **matherr**, a non-zero return indicates that the proposed return value in the exception vector has been changed and that the new value should be used. A zero return indicates that the proposed return value is OK.

For **except**, the actual return value (a double) is passed back.

SEE ALSO

_fperr

NAME

memory	Memory block operations
memccpy	Copy a memory block up to a char
memchr	Find a character in a memory block
memcmp	Compare two memory blocks
memcpy	Copy a memory block
memset	Set a memory block to a value
movmem	Move a memory block
repmem	Replicate values through a block
setmem	Set a memory block to a value
swmem	Swap two memory blocks

SYNOPSIS

```
#include <string.h>

s = memccpy(to,from,c,n);
s = memchr(a,c,n);
x = memcmp(a,b,n);
s = memcpy(to,from,n);
s = memset(to,c,n);

movmem(from,to,n);
repmem(to,vt,nv,nt);
setmem(to,n,c);
swmem(a,b,n);
```

```
char *to; destination pointer
char *from; source pointer
unsigned n; number of bytes
char c; character value
char *a,*b; block pointers
char *vt; value template
int nv; number of bytes in template
int nt; number of templates in block
char *s; return pointer
int x; return value
```

DESCRIPTION

These functions manipulate blocks of memory in various ways.

The **memcpy** and **movmem** functions are similar, except the former was introduced with UNIX V, while the latter is a traditional Lattice function. In a like manner, **memset** and **setmem** perform the same operation, except that the former is UNIX-compatible. Note that **memcpy** and **memset** return a pointer to the destination block, while **movmem** and **setmem** have void returns. Also note that **memcpy** and **movmem** are smart enough to handle overlapping memory blocks correctly.

The **memccpy** function is similar to **memcpy** except that copying stops after the specified block size has been copied or after the specified character has been copied. It returns a pointer to the character after **c** in the **from** block, or a null pointer if **c** was not found in the first **n** characters. Unlike **memcpy**, **memccpy** does not handle overlapping memory blocks. If you specify overlapping blocks to this function, the results are unpredictable.

The **memchr** function returns a pointer to the first occurrence of the specified character in the block, or a null pointer if the character is not found.

The **memcmp** function performs a character-by-character comparison of two memory blocks and returns an integral value as follows:

NEGATIVE = > first block is below second

ZERO = > first block equals second

POSITIVE = > first block is above second

There is currently no UNIX equivalent for **swmem** and **repmem**. The former merely swaps two blocks in memory, although it has a major performance advantage over the typical for-loop approach. The latter replicates a template of values throughout a block and is very useful when you need to initialize an array of structures to some non-zero pattern.

RETURNS

As noted above.

> NOTE: These functions neither recognize nor produce the null terminator byte usually found at the end of strings. A popular mistake is to assume that **memcpy**, in common with **strcpy**, automatically places a null byte at the end of the block. It does not.

NAME

mkdir Create a new directory

SYNOPSIS

```
#include <stdio.h>

error = mkdir(path);

int error; 0 if successful
char *path; points to new directory path string
```

DESCRIPTION

This function makes a new directory in the specified path. For example, if **path** is *sys:/abc/def/ghi*, then the new directory is named *ghi* and is in the path */abc/def* on the system drive. For AmigaDOS, the path may begin with a drive or volume name and a colon.

RETURNS

If the operation is successful, the function returns 0. Otherwise it returns -1 and places error information in **errno** and **_OSERR**.

SEE ALSO

errno, _OSERR

NAME

onbreak Plant break trap

SYNOPSIS

```
#include <dos.h>

error = onbreak(func);

int error; 0 if successful
int (*func)(); pointer to function to be called
```

DESCRIPTION

This function plants a break trap, which is simply a function that gets called whenever the user keys CTRL-C or CTRL-D. The function can perform any AmigaDOS operations. If it returns a value of 0, then execution resumes at the interrupted point. Otherwise, the program is aborted immediately.

If **func** is null, then the current break trap, if any, is removed and the default interrupt handler is restored. With the default handler, CTRL-C and CTRL-D cause a requester to appear on the screen with choices of **continue** or **abort**.

Detection of CTRL-C and CTRL-D is performed during level 1 I/O. Explicit checks for these events can be forced by calls to the function **chkabort**.

RETURNS

The **onbreak** function returns 0 if it was successful. The break trap function should return 0 to continue execution and non-zero to abort.

SEE ALSO

chkabort

EXAMPLES

```
/*
*
* This program tests the onbreak function.  After
* the initial message is printed, you should get
* the Break received message when you hit CTRL C
* or CTRL D. The second time will cause the program
* to terminate.
*
*/
#include <stdio.h>
int i = 0;

int brk()  /* This is the break function */
{
printf("Break received...\n");
return(i++);
}

main()   /* This is the main program */
{
printf("Setting break trap...\n");
if(onbreak(&brk)) printf("Can't set break trap\n");
while(1) chkabort(); /* check for CTRL-C */
}
```

onexit

Exit trap *Class: ANSI*

NAME

onexit Exit trap

SYNOPSIS

```
#include <stdlib.h>

success = onexit(func);

int success; non-zero if successful
int (*func)(); pointer to trap function
```

DESCRIPTION

This function establishes a *trap* which will be called when the program terminates. The trap function is called just before the program returns to the operating system. For normal termination via the **exit** function or via a return from the **main** function, all buffers are flushed and files are closed before the trap is called. If the program is using **_exit**, the files and buffers may still be open, depending on what the program does before terminating. In both cases, user-allocated memory is not yet freed.

As an extension to the ANSI definition of this function, we pass the exit code to the trap function as its only argument. Then whatever value the trap function returns is used as the real exit code.

Another difference between our implementation and the ANSI definition is that we allow only one trap. Each call to **onexit** overrides the previous trap. If you call **onexit** with a null pointer, the current trap is removed.

> NOTE: The exit trap is called after all files have been closed. A common mistake is to issue some type of output message via **printf** or **cprintf** from within the exit trap. In order for this to work, you should **fopen** or **open** the console device and send the message via **fprintf** or **write**.

SEE ALSO

exit, _exit

EXAMPLES

```
/*
*
* This program tests the "onexit" function.
*
*/
#include <stdio.h>

int ex(i) /* This is the exit trap function */
int i;
{
FILE *con;

if((con = fopen("*","w")) != NULL)
  {
  fprintf(con,"Exit trap hit...code %d found\n",i);
  fclose(con);
  }
return(0);
}

main()  /* This tests the exit trap */
{
int (*p)();
p = &ex;
printf("Setting exit trap...\n");
if(!onexit(p)) printf("Can't set trap...\n");
printf("Exiting with code 2\n");
exit(2);
}
```

NAME

nofill Open a Level 1 file

SYNOPSIS

```
#include <fcntl.h>

fh = open(name,mode,prot);

int fh; file handle
char *name; file name
int mode; access mode
int prot; protection mode (O_CREAT only)
```

DESCRIPTION

This function opens a file so that it can be accessed via the level 1 I/O functions. The file name can be any character string that is a valid file name, and it may include a device code and a directory path. The access mode is formed by ORing together the appropriate symbols from the following list:

CONVENTIONAL UNIX SYMBOLS...

O_RDONLY	Read-only access. No writes are allowed.
O_WRONLY	Write-only access. No reads are allowed.
O_RDWR	Read-write access. Both reads and writes are allowed.
O_NDELAY	This symbol is defined for UNIX compatibility and has no effect under AmigaDOS.
O_APPEND	This symbol is normally used in conjunction with O_WRONLY or O_RDWR. It causes the I/O system to seek

to the end of the file before each write operation. After each write operation, the file is positioned at the new end-of-file.

O_TRUNC — If the file exists, it is truncated to a length of 0. This flag is normally used with O_CREAT, O_WRONLY or O_RDWR.

O_CREAT — If the file does not already exist, it is created. The protection mode argument is provided for compatibility with existing software, but is ignored under AmigaDOS. To set the protection use the **chmod** function to change the protection bits after the file has been closed.

O_EXCL — This symbol is used only with O_CREAT. If O_EXCL and O_CREAT are both present and the file already exists, the **open** function will fail.

RETURNS

If the operation is successful, the function returns a file handle, which is an integer equal to or greater than 0. Otherwise it returns -1 and places error information in **errno** and **_OSERR**.

SEE ALSO

errno, _OSERR, chgfa, chmod, close, creat

NAME

perror Print UNIX error message

SYNOPSIS

```
#include <stdio.h>

error = perror(s);

int error; contents of errno
char *s; message prefix
```

DESCRIPTION

This function checks **errno** and, if it is non-zero, sends an error message to **stderr**. The message consists of the specified prefix, a colon and space, and the message text from the external array named **sys_errlist**. This array contains pointers to the various UNIX error messages. The highest error number is given by the contents of external integer **sys_nerr**. The Lattice compiler package contains the source for these two external items in a file named **syserr** which allows you modify the messages as you desire.

RETURNS

The function returns the contents of **errno** so you can test for an error condition and print a message in one step, as in the example:

```
if(perror("foo")) goto abort;
```

Note that this feature is an extension of UNIX and if you use it, your program may be non-portable.

Class: ANSI *Print UNIX error message*

SEE ALSO

errno, sys_nerr, sys_errlist, poserr

poserr

Print AmigaDOS error message *Class: AMIGA*

NAME

poserr Print AmigaDOS error message

SYNOPSIS

```
#include <dos.h>

error = poserr(s);

int error; contents of _OSERR
char *s; message prefix
```

DESCRIPTION

This function checks **_OSERR** and, if it is non-zero, sends an error message to **stderr.** The message consists of the specified prefix, a colon and space, and the message text from the external array named **os_errlist**. This array of structures contains pointers to the various AmigaDOS error messages. The highest error number is given by the contents of external integer **os_nerr**. The Lattice compiler package contains the source for these two external items in a file named **oserr.c** which you can modify to customize or expand the messages.

RETURNS

The function returns the contents of **_OSERR** so you can test for an error condition and print a message in one step, as in the example:

```
if(poserr("foo")) goto abort;
```

SEE ALSO

_OSERR, os_errlist, os_nerr, perror

NAME

qsort	Sort a data array
qsort	Sort a data array
dqsort	Sort an array of doubles
fqsort	Sort an array of floats
lqsort	Sort an array of long integers
sqsort	Sort an array of short integers
tqsort	Sort an array of text pointers

SYNOPSIS

```
#include <stdlib.h>

qsort(a,n,size,cmp);
dqsort(da,n);
fqsort(fa,n);
lqsort(la,n);
sqsort(sa,n);
tqsort(ta,n);

char *a; data array pointer
double *da; pointer to double array
float *fa; pointer to float array
long *la; pointer to long int array
short *sa; pointer to short int array
char *ta[ ]; pointer to text pointer array

int n; number of elements in array
int size; element size in bytes
int (*cmp)(); pointer to comparison function
```

DESCRIPTION

The **qsort** function sorts the specified data array using the ACM 271 algorithm, more popularly known as *Quicksort*. During its operation, it calls upon the specified comparison routine with pointers to the two array elements being compared. As an extension to the normal UNIX implementation, we also pass a third argument, the element size, which can be ignored. The comparison routine should return an integral result as follows:

NEGATIVE = >	first element is below second
POSITIVE = >	first element is above second
ZERO = >	elements are equal

The **dqsort, fqsort, lqsort, sqsort** and **tqsort** functions sort various arrays which are commonly encountered. They are all straightforward except for **tqsort**, which requires some explanation. The **ta** array consists of pointers to null-terminated character strings. The **tqsort** function re-arranges the pointers so that the strings are in ascending ASCII sequence, using **strcmp** as the comparison routine. Note that it is not the pointer values that are being sorted, but the strings to which the pointers refer.

NAME

rand, srand	Simple random number generator
rand	Generate a random number
srand	Set seed for rand function

SYNOPSIS

```
#include <stdlib.h>

x = rand();
srand(seed);

int x; random number
unsigned seed; random number seed
```

DESCRIPTION

The **rand** function returns pseudo-random numbers in the range from 0 to the maximum positive integer value. When you call **srand**, the random number generator is reset to a new seed value. The initial default seed is 1.

See **drand48** and its related functions for more sophisticated random number generation.

RETURNS

As noted above.

Class: ANSI *Simple random number generator*

SEE ALSO

drand48

EXAMPLES

```
/*
*
* This example prints 1000 random numbers
*
*/
#include <stdio.h>
#include <stdlib.h>

main(argc,argv)
int argc;
char *argv[ ];
{
int i;
unsigned x;

if(argc > 1)
  {
  stcd_u(argv[1],&x);
  if(x == 0) x = 1;
  printf("Seed value is %d\n",x);
  srand(x);
  }
printf("Here are 1000 random numbers...\n");
for(i = 0; i < 200; i++)
  printf("%5d %5d %5d %5d %5d\n",
         rand(),rand(),rand(),rand(),rand());
printf("\n\n");
}
```

read, write

Read or write a Level 1 file *Class: UNIX*

NAME

read, write	Read or write a Level 1 file
read	Read from Level 1 file
write	Write to Level 1 file

SYNOPSIS

```
#include <fcntl.h>

count = read(fh,buffer,length);
count = write(fh,buffer,length);

int count; actual bytes read or written
int fh; file handle
char *buffer; data buffer
int length; number of bytes to read or write
```

DESCRIPTION

These functions read or write a Level 1 file whose handle was returned by **creat** or **open**. Under normal circumstances, the value returned should match the buffer length. If this value is -1 or greater than the requested length, then some type of error occurred, and you should consult **errno** and **_OSERR**. If the actual length is less than the requested length when reading, this usually means that the file is exhausted. Similarly, if the actual length is less than the requested length for a write operation, this usually means that the device has no more space available. In both of these cases, it is still a good idea to check **errno** and **_OSERR** just in case some malfunction caused the short count.

Note that these functions are very similar to the Level 0 I/O functions **dread** and **dwrite**. The primary difference is that Level 1 files will be automatically closed by **exit** and **_exit**, which are usually called for you when the program terminates.

RETURNS

If the operation is successful, the function returns the actual number of bytes transferred. Otherwise it returns -1 and places error information in **errno** and **_OSERR**.

SEE ALSO

errno, _OSERR, open, dread, dwrite

NAME

remove, unlink	Remove a file
remove	Remove a file
unlink	Remove a file

SYNOPSIS

```
#include <stdio.h>

error = remove(name);
error = unlink(name);

int error; non-zero if error
char *name; file name
```

DESCRIPTION

These functions remove the specified file from the system. They behave identically, but **unlink** is provided for compatibility with some versions of UNIX. The **remove** function is preferred because it is now in the ANSI C standard.

The file name argument can include a path, but it cannot include wild card characters. That is, you can remove only one file at a time.

RETURNS

If a non-zero value is returned, some type of error occurred, and additional information can be found in **errno** and **_OSERR**. The most common errors occur when you try to remove a file that doesn't exist or that is marked as read-only.

SEE ALSO

errno, _OSERR

EXAMPLES

```
/*
*
* This program removes all files specified in the
* argument list.   It does not allow wild card
* characters in the file names.
*
*/
#include <stdio.h>
main(argc,argv)
int argc;
char *argv[ ];
{
int i;  /* loop counter */
int ret = 0; /* exit code, non-zero if any failures */

for(i = 1; i < argc; i++) if(remove(argv[i])
  {
  perror("RMV");
  ret = 1;
  }
exit(ret);
}
```

NAME

rename Rename a file

SYNOPSIS

```
#include <stdio.h>

error = rename(old,new);

int error; 0 for success, -1 for error
char *old; old file name
char *new; new file name
```

DESCRIPTION

This function renames a file, if possible. The old name can include a path, but the new name should not. A failure occurs if the old file cannot be found or if the new name is the same as an existing file.

RETURNS

If the function fails, it returns -1 and places additional error information into **errno** and **_OSERR**. Success is indicated by a return value of 0.

EXAMPLES

```
/*
*
* This is a version of the rename command
* that prompts for the old and new names.
*
*/
#include <stdlib.h>
#include <stdio.h>
#include <dos.h>

main(argc,argv)
```

```
int argc;
char *argv[ ];
{
char old[FMSIZE],new[FMSIZE];
char *pold,*pnew;

if(argc < 2)  /* Get old file name */
  {
  printfOLD FILE: );
  if(gets(old) == NULL) exit(1);
  pold = old;
  }
else pold = argv[1];

if(arc < 3)
  {
  printf("NEW FILE: ");
  if(gets(new) == NULL) exit(1);
  pnew = new;
  }
else pnew = argv[2];

if(rename(pold,pnew))
  {
  perror("RENAME");
  exit(1);
  }
}
```

NAME

rlsmem, rlsml	Release a Level 2 memory block
rlsmem	Release a Level 2 memory block
rlsml	Release a Level 2 memory block

SYNOPSIS

```
#include <stdlib.h>

error = rlsmem(p,sbytes);
error = rlsml(p,lbytes);

int error; non-zero if error
char *p; block pointer
short sbytes; number of bytes
long lbytes; number of bytes
```

DESCRIPTION

These functions release memory blocks that were previously obtained via **getmem** or **getml**. If the number of bytes is less than the value **MELTSIZE** as defined in header file **dos.h**, then it is made equal to **MELTSIZE**.

Note that you can free a portion of a block as long as the portion being freed is at least **MELTSIZE** bytes long. For example, suppose you allocate a block of 100 bytes, use the lower 80, and decide to return the upper 20 to the memory pool. Simply make the following call:

```
rlsmem(p+80,20);
```

where **p** is the original block pointer. Again, whenever you free partial blocks in this way, make sure that the portion being freed is equal to or greater than **MELTSIZE**.

RETURNS

If the block is not in the current Level 2 memory pool or overlaps a block that is already free, a value of -1 is returned. Otherwise, the return value is 0.

EXAMPLES

```
/*
*
* This program builds a linked list of text strings
* obtained from the standard input file.
*
*/
#include <stdio.h>
#include <stdlib.h>
/*
*
* These elements are linked together to form the
* text string list.
*
*/
#define MAX 256
struct LIST
  {
  struct LIST *next; /* forward linkage */
  int size;  /* element size */
  char text[MAX]; /* minimum text string */
  };
/*
*
* Main program
*
*/
main()
{
struct LIST *p,*q,*list = NULL;
char b[MAX];
int x;
/*
*
* Build the list
*
*/
for(q = (struct LIST *)(&list);;q = p)
```

```
    {
    printf("Enter a text string: ");
    if(gets(b) == NULL) break;
    if(b[0] == '\0') break;
    p = (struct LIST *)getmem(sizeof(struct LIST));
    if(p == NULL)
      {
      printf("No more memory\n");
      break;
      }
    p->next = q->next;
    x = MAX - (stccpy(p->text,b,256);
    p->size = sizeof(struct LIST) - x;
    if(x >= MELTSIZE) rlsmem(p+p->size,x);
    }
/*
*
* Print the list
*
*/
printf("\n\nTEXT LIST...\n");
for(p = list; p != NULL; p = p->next)
   printf("%s",p->text);
}
```

NAME

rmdir Remove a directory

SYNOPSIS

```
#include <stdio.h>

error = rmdir(path);

int error; 0 if successful
char *path; points to directory path string
```

DESCRIPTION

This function removes an existing directory in the specified path. For example, if **path** is *sys:/abc/def/ghi*, then the directory named *ghi* is removed from the path */abc/def* on the system drive. For AmigaDOS, the path may begin with a drive or volume name and a colon.

RETURNS

If the operation is successful, the function returns 0. Otherwise it returns -1 and places error information in **errno** and **_OSERR**.

SEE ALSO

errno, _OSERR

setbuf

Set buffer mode for a Level 2 file *Class: ANSI*

NAME

setbuf	Set buffer mode for a Level 2 file
setnbf	Set non-buffer mode for L2 file
setvbuf	Set variable buffer for L2 file

SYNOPSIS

```
#include <stdio.h>

setbuf(fp,buff);
setnbf(fp);
error = setvbuf(fp,buff,type,size);

int error; 0 if successful
FILE *fp; file pointer
char *buff; buffer pointer
int type; type of buffering
int size; buffer size in bytes
```

DESCRIPTION

These functions set the buffering mode for a Level 2 file. Proper usage is to call the appropriate function after calling **fopen** and before calling any other Level 2 I/O functions. If you fail to follow this rule, the file may become corrupted.

The Level 2 I/O system automatically allocates a buffer via **getmem** when you perform the first read or write operation. Then the data being read or written is staged through this buffer in order to improve I/O efficiency. If you would rather use your own buffer instead of having one allocated for you, call **setbuf** with a non-null buffer pointer. The buffer size must be at least as large as the value given in the external integer **_bufsiz**, which defaults to the value of the symbol **BUFSIZ**, defined in **stdio.h**.

You can eliminate buffered I/O by calling **setnbf** or by calling **setbuf** with a null buffer

pointer. When this is done, physical I/O occurs whenever your program performs Level 2 read or write operation, even if only one byte is being transferred. This is very inefficient for disk files but often desirable for terminal or communication ports.

The **setvbuf** function can do everything that the other two functions can do, and it can also set *line buffered* mode and attach a buffer of non-standard size. The **type** argument must be one of the following symbols defined in **stdio.h**:

_IOFBF = > Fully buffered

_IOLBF = > Line buffered

_IONBF = > Non-buffered

For **_IOFBF** and **_IOLBF**, the specified buffer will be attached to the file unless **buff** is null, in which case a buffer will be automatically allocated on the first read or write. For the **_IONBF** case, the **buff** and **size** arguments are ignored.

The line-buffered mode is useful for interactive applications. When in this mode, the buffer is flushed whenever a newline is sent, the buffer is full, or input is requested. Note, however, that you must use the **fputc** and **fputchar** functions instead of the **putc** and **putchar** macros in order for line buffering to work correctly. The macros do not check if line-buffered mode is active, and so they behave as if the file were fully buffered.

RETURNS

For **setvbuf**, the error code is non-zero if **type** or **size** is invalid.

> NOTE: These functions must be used only after **fopen** and before any other Level 2 file operations. A common error is to allocate a buffer on the stack within a function, attach it to a file, and then return from the function. This will corrupt the stack and cause a system failure.

SEE ALSO

fopen

NAME

signal Establish event traps

SYNOPSIS

```
#include <signal.h>

oldfun = signal(sig,newfun);

int (*oldfun)(); old trap function
int sig; signal number
int (*newfun)(); new trap function
```

DESCRIPTION

This function establish traps for various events that can occur outside of your program.

The **newfun** argument specifies the action to be taken when the signal occurs, as follows:

SIG_IGN	Ignore the signal.
SIG_DFL	Take the system default action, as indicated above for each signal.

If **newfun** is not any of the above, then it must be a valid function pointer. When the signal is detected, the action is reset to either SIG_DFL or SIG_IGN, depending on the particular signal. Then the trap function is called with an integer argument specifying which signal was detected (e.g. SIGINT). The trap function can take whatever action is necessary, including calling **signal** again to re-establish itself as the trap function. If the function returns, execution continues at the point in your program where the signal was detected.

The **sig** argument specifies which signal is being trapped, using the following symbols defined in **signal.h**:

SIGFPE — This signal occurs whenever a floating point error is detected and the standard version of **CXFERR** is installed. If you install your own version, you must duplicate our code (supplied as a file named *CXFERR.C*) in order to provide this signal.

SIGINT — This signal occurs whenever the user operates the CTRL-C or CTRL-D key combination. The default action for AmigaDOS is to abort your program. If you specify a function to be called, the signal will be reset to SIG_IGN when the interrupt occurs. Your function should call **signal** again if you want to re-install the trap.

RETURNS

The **signal** function normally returns the previous value of the trap function, which may be SIG_IGN or SIG_DFL.

NAME

sizmem Get Level 2 memory pool size

SYNOPSIS

```
#include <stdlib.h>

size = sizmem();

long size;
```

DESCRIPTION

This function returns the number of unallocated bytes in the current Level 2 memory pool. This value is the sum of the sizes of all unallocated blocks, and so it does not indicate the size of the largest free block.

Also, the value does not indicate the maximum amount of Level 2 memory that can be allocated. That is, the Level 2 allocation functions **getmem** and **getml** will automatically expand the pool via **sbrk** when no block of sufficient size is found in the pool.

SEE ALSO

getmem, getml, rlsmem, rlsml, rstmem

NAME

stcarg Get an argument

SYNOPSIS

```
#include <string.h>

length = stcarg(s,b);

int length; number of bytes in argument
char *s; text string pointer
char *b; break string pointer
```

DESCRIPTION

This function scans the text string until one of the break characters is found or until the null terminating byte is hit. While scanning, **stcarg** skips over substrings that are enclosed in single or double quotes, and the backslash is recognized as an escape character. In other words, break characters will not be detected if they are quoted or preceded by a backslash.

RETURNS

The function returns a count of the number of characters in s up to but not including the break character or null terminator.

SEE ALSO

stpbrk, strcspn, strpbrk

EXAMPLES

```
#include <stdio.h>
#include <string.h>

main()
{
char a[256],b[256];
int x;

while(1)
  {
  printf("Enter text string:   ");
  if(gets(a) == NULL) exit(0);
  printf("Enter break string: ");
  if(gets(b) == NULL) exit(0);
  x = stcarg(a,b);
  printf("Arg length: %d, Arg text: %.*s\n",x,x,a);
  }
}
```

stccpy, stpcpy, strcpy, strncpy

Copy one string to another *Class: ANSI*

NAME

stccpy, stpcpy, strcpy, strncpy	Copy one string to another
stccpy	Copy one string to another
stpcpy	Copy one string to another
strcpy	Copy one string to another
strncpy	Copy string, length-limited

SYNOPSIS

```
#include <string.h>

size = stccpy(to,from,n)
np = stpcpy(to,from);
p = strcpy(to,from);
p = strncpy(to,from,n);

char *np; points to end of destination string
char *p; same as destination pointer
char *to; destination pointer
char *from; source pointer
int n; maximum source length
```

DESCRIPTION

These functions copy the null-terminated source string to the destination area. For **stpcpy** and **strcpy**, the entire source string is copied, and the resulting destination is always null-terminated. The **strncpy** function always writes exactly **n** characters to the destination. If the null terminator is hit before **n** characters are copied from the source, then the destination is filled with null bytes. If the source string contains more than **n** non-null characters, the destination will not be null-terminated.

Class: ANSI *Copy one string to another*

The **stccpy** function is similar to **strncpy** except that it always produces a null-terminated string, and it returns actual number of bytes placed in the **to** area, including the null terminator.

RETURNS

The **strcpy** and **strncpy** functions return a pointer that is the same as the destination pointer. The Lattice function **stpcpy** returns a pointer to the end of the destination string, which is often more useful when you are building a string up from several pieces.

> NOTE: Be careful when using **strncpy**, since it is one of the few string functions which does not produce a null-terminated string under every condition.

EXAMPLES

```
/*
*
* This example should print: Hello, my name is Flo.
*
*/
#include <string.h>

main()
{
char b[256],*p;

p = stpcpy(b,"Hello, ");
p = stpcpy(p,"my name ");
p = stpcpy(p,"Flo.");
puts(b);
}
```

NAME

stcd_i, et al	Convert strings to integer
stcd_i	Convert decimal string to int
stco_i	Convert octal string to int
stch_i	Convert hexadecimal string to int
stcd_l	Convert decimal string to long int
stco_l	Convert octal string to long int
stch_l	Convert hexadecimal string to long

SYNOPSIS

```
#include <string.h>

length = stcd_i(in,ivalue);
length = stci_o(in,ivalue);
length = stci_h(in,ivalue);
length = stcl_d(in,lvalue);
length = stcl_o(in,lvalue);
length = stcl_h(in,lvalue);

int length; input length
char *in; input string pointer
int *ivalue; integer value pointer
long *lvalue; long integer value pointer
```

DESCRIPTION

These functions scan an input string and convert the leading characters into short or long integers. For **stcd_i** and **stcd_l**, the input string must begin with a plus sign '+', minus sign '-', or a decimal digit ('0' to '9'). The octal conversions **stco_i** and **stco_l** process an unsigned string of octal digits ('0' to '7'). Finally, the hexadecimal conversions **stch_i** and **stch_l** handle unsigned strings containing digits from '0' to '9' and letters from 'A' to 'F' or 'a' to 'f'. Scanning of the input string stops when the first invalid character is reached, At that point, the resulting value is stored into the area addressed by the second argument.

RETURNS

Each function returns the number of input characters converted. This result will be 0 if the first character of the input string is not valid for the particular conversion. In that case, conversion result stored via the second argument will be 0.

EXAMPLES

```
#include <stdio.h>
#include <string.h>

main()
{
int x;
long j;
char b[80];

while(1)
   {
   printf("\nEnter a hexadecimal value: ");
   if(gets(b) == NULL) exit(0);
   x = stch_l(b,&j);
   printf("stch_l: Length %d, Result %lx\n",x,j);
   }
}
```

NAME

stcgfe, stcgfn, stcgfp	Get file name components
stcgfe	Get file extension
stcgfn	Get file node
stcgfp	Get file path

SYNOPSIS

```
#include <string.h>

size = stcgfe(ext,name);
size = stcgfn(node,name);
size = stcgfp(path,name);
int size; size of result string
char *ext; extension area pointer
char *node; node area pointer
char *path; path area pointer
char *name; file name pointer
```

DESCRIPTION

These functions isolate the path, node, or extension portion of a file name. The node is the rightmost portion of the file name that is separated from the rest of the name by a colon, slash, or backslash. The extension is the final part of the node that begins with a period, and the path is the leading part of the name up to the node.

For example:

NAME	PATH	NODE	EXTENSION
myprog.c	-	myprog.c	c
/abc.dir/def	/abc.dir	def	-
/abc.dir/def.ghi	/abc.dir	def.ghi	ghi
df0:yourfile	df0:	yourfile	-
/abc/	/abc	-	-

RETURNS

The **size** value is the same as would be returned by the **strlen** function. That is, if **size** is 0, then the desired portion of the file name could not be found and the result area contains a null string.

SEE ALSO

strsfn

NAME

stcis, stcisn, strspn, strcspn	Measure character set span
stcis	Measure span of chars in set
stcisn	Measure span of chars not in set
strspn	Measure span of chars in set
strcspn	Measure span of chars not in set

SYNOPSIS

```
#include <string.h>

length = stcis(s,b);
length = stcisn(s,b);
length = strspn(s,b);
length = strcspn(s,b);

int length; span length in bytes
char *s; points to string being scanned
char *b; points to character set string
```

DESCRIPTION

These functions measure the number of characters at the beginning of input string **s** that are either in or not in the character set specified by **b**. The **stcis** and **strspn** functions are identical and count the number of leading characters that are in the set. Similarly, **stcisn** and **strcspn** are identical and count the number of leading characters that are not in the set. The *stc* pair are provided for compatibility with other version of Lattice C, while the *str* functions are now part of the ANSI standard.

Class: ANSI *Measure character set span*

RETURNS

The functions all return the number of bytes that are in or not in the specified character set. Note that the scan always stops when the null terminator byte is reached.

EXAMPLES

```
#include <stdio.h>
#include <string.h>

main()
{
char s1[256],s2[256];

while(1)
        {
        printf("\nEnter test string: ");
        if(gets(s1) == NULL) exit(0);
        printf("Enter span string: ");
        if(gets(s2) == NULL) exit(0);
        printf("strspn: %d\n",strspn(s1,s2));
        printf("strcspn: %d\n",strcspn(s1,s2));
        printf("stcis: %d\n",stcis(s1,s2));
        printf("stcisn: %d\n",stcisn(s1,s2));
        }
}
```

NAME

stci_d, et al	Convert integers to strings
stci_d	Convert int to decimal
stci_o	Convert int to octal
stci_h	Convert int to hexadecimal
stcl_d	Convert long int to decimal
stcl_o	Convert long int to octal
stcl_h	Convert long int to hexadecimal
stcu_d	Convert unsigned int to decimal
stcul_d	Convert unsigned long to decimal

SYNOPSIS

```
#include <string.h>

length = stci_d(out,ivalue);
length = stci_o(out,ivalue);
length = stci_h(out,ivalue);
length = stcl_d(out,lvalue);
length = stcl_o(out,lvalue);
length = stcl_h(out,lvalue);
length = stcu_d(out,uivalue);
length = stcul_d(out,ulvalue);

int length; output length
char *out; output buffer pointer
int ivalue; integer value
long lvalue; long integer value
unsigned int uivalue; unsigned integer value
```

```
unsigned long ulvalue; unsigned long integer value
```

DESCRIPTION

These functions convert various integral values into ASCII strings. The output area must be large enough to accomodate the maximum possible string, including the terminating null byte that each function appends. The following table shows the required lengths.

FUNCTION	LENGTH
stci_d	07
stci_o	07
stci_h	05
stcl_d	13
stcl_o	12
stcl_h	09
stcu_d	06
stcul_d	12

For **stci_d** and **stcl_d**, the first output character will be a minus sign if the input value is negative. No special leading character is generated if the value is positive. For all functions, leading zeroes are suppressed, and a single '0' character is generated if the input value is 0.

RETURNS

The return value is the number of characters actually placed into the output area, not including the final null byte.

EXAMPLES

```
#include <stdio.h>
#include <string.h>

main()
{
int i,x;
char b[13];

while(1)
        {
        printf("\nEnter a short integer: ");
        scanf("%d",&i);
        x = stci_d(b,i);
        printf("stci_d: Length %d, Result %s\n",x,b);
        x = stci_o(b,i);
        printf("stci_o: Length %d, Result %s\n",x,b);
        x = stci_h(b,i);
        printf("stci_h: Length %d, Result %s\n",x,b);
        }
}
```

NAME

stcpm, stcpma	Pattern match functions
stcpm	Un-anchored pattern match
stcpma	Anchored patter match

SYNOPSIS

```
#include <string.h>

size = stcpm(string,pattern,match);
size = stcpma(string,pattern);

int size; size of matching string
char *string; string to be scanned
char *pattern; pattern string
char **match; returns pointer to matching string
```

DESCRIPTION

These functions scan a string to find a specified pattern. The pattern is specified in a simplified form of regular expression notation where

?	matches any single character
c*	matches zero or more occurrences of character c
c+	matches one or more occurrences of character c
\?	matches a question mark (?)
*	matches an asterisk (*)
\+	matches a plus sign (+)

Any other character must match exactly. Some examples are:

PATTERN	MATCHES
abc	Only *abc*
*ab*c*	*ac*, *abc* or *abbc* etc.
ab+c	*abc*, *abbc* or *abbbc* etc.
*ab?*c*	A string starting with *ab* and ending in *c*, e.g.*abxyzc*
*ab*c*	Only *ab*c*

For **stcpma**, the match must occur at the beginning of the string, while for **stcpm**, the match can occur anywhere in the string. In either case, the function returns the size of the matching string or zero if there was no match. Also, **stcpm** returns a pointer to the beginning of the matching string.

EXAMPLES

```
#include <stdio.h>
#include <string.h>

main()
{
char s[100],p[100],*r;
int x;

while(1)
  {
  printf("\nSearch string => ");
  if(gets(s) == NULL) break;
  printf("Pattern       => ");
  if(gets(p) == NULL) break;
  x = stcpma(s,p);
  if(x)
    printf("stcpma: size = %d, match = \"%.*s\"\n",x,x,s);
  else
    printf("stcpma: no match\n");
  x = stcpm(s,p,&r);
  if(x)
    printf("stcpm: size = %d, match = \"%.*s\"\n",x,x,r);
  else
    printf("stcpm: no match\n");
```

```
        }
    }
```

NAME

stpblk Skip blanks (white space)

SYNOPSIS

```
#include <string.h>

q = stpblk(p);

char *q; updated string pointer
char *p; string pointer
```

DESCRIPTION

This function advances the string pointer past *white space* characters, that is, past all the characters for which **isspace** is true.

RETURNS

The function returns a pointer to the next non *white space* character. Note that the null terminator byte is not considered to be white space, and so the function will not go past the end of the string.

SEE ALSO

stcis, strspn

EXAMPLES

```
#include <stdio.h>
#include <string.h>

main()
{
char input[256];

while(1)
    {
    puts("\nEnter a string with leading blanks...");
    if(gets(input) == NULL) exit(0);
    printf("%s\n",stpblk(input));
    }
}
```

NAME

stpbrk, strpbrk	Find break character in string
stpbrk	Find break character in string
strpbrk	Find break character in string

SYNOPSIS

```
#include <string.h>

p = stpbrk(s,b);
p = strpbrk(s,b);

char *p; points to break character in s
char *s; string to be scanned
char *b; break characters
```

DESCRIPTION

These functions scan string **s** to find the first occurrence of a character from break string **b**. They are completely equivalent, except that **strpbrk** is the UNIX name, while **stpbrk** is the traditional Lattice name.

RETURNS

If no character from **b** is found in **s**, a NULL pointer is returned. Otherwise, **p** is a pointer to the break first break character.

SEE ALSO

strspn, strcspn

EXAMPLES

```
#include <string.h>

/*
*
* Scan for commas, periods, and blanks.  Display
* the tail of the string each time a break
* character is found.
*
*/
char *p,s[ ] = "Hello, my name is Herkimer Q. Jerkimer."

for(p = s; (p = strspn(p,",. ")) != NULL;
    printf("%s\n",p));
```

NAME

stpchr, stpchrn, strchr, strrchr	Find character in string
stpchr	Find character in string
stpchrn	Find character not in string
strchr	Find character in string
strrchr	Find character not in string

SYNOPSIS

```
#include <string.h>

p = stpchr(s,c);
p = stpchrn(s,c);
p = strchr(s,c);
p = strrchr(s,c);

char *p; updated string pointer
char *s; input string pointer
char c; character to be located
```

DESCRIPTION

The **stpchr** and **strchr** functions scan the input string to find the first occurrence of the character specified by argument **c**. Similarly, **stpchrn** and **strrchr** scan for the first occurrence of some character other than **c**. The *stp* versions are provided for compatibility with other versions of Lattice C, while the *str* functions are now part of the ANSI standard.

Class: ANSI *Find character in string*

RETURNS

For **stpchr** and **strchr**, a NULL pointer is returned if the input string is empty or if the specified character is not found. The other two functions return a NULL pointer if the input string is empty or consists entirely of character **c**.

EXAMPLES

```
#include <stdio.h>
#include <string.h>

main()
{
char c,s[256];

while(1)
        {
        printf("\nEnter test string: ");
        if(gets(s1) == NULL) exit(0);
        printf("Enter character: ");
        if((c = getchar()) == EOF) exit(0);
        printf("stpchr: %s\n",stpchr(s,c));
        printf("stpchrn: %s\n",strcspn(s,c));
        printf("strchr: %s\n",stcis(s,c));
        printf("strrchr: %s\n",stcisn(s,c));
        }
}
```

NAME

stpdate Convert date array to string

SYNOPSIS

```
#include <string.h>

np = stpdate(p,mode,date);

char *np; updated output string pointer
char *p; output string pointer
int mode; conversion mode
char *date; date array, as follows
date[0] => year - 1980
date[1] => month (1 to 12)
date[2] => day (1 to 31)
```

DESCRIPTION

This function converts a 3-byte date array into ASCII or BCD according to the **mode** argument:

0 = >	yymmdd (BCD, 3 bytes)
1 = >	yymmdd (ASCII, 7 bytes)
2 = >	mm/dd/yy (ASCII, 9 bytes)
3 = >	mm-dd-yy (ASCII, 9 bytes)
4 = >	MMM d, yyyy (ASCII, up to 13 bytes)
5 = >	Mm...m d, yyyy (ASCII, up to 19 bytes)
6 = >	dd MMM yy (ASCII, 10 bytes)
7 = >	dd MMM yyyy (ASCII, 12 bytes)

In the above formats, *MMM* represents a 3-character month abbreviation in capitals, and *Mm...m* represents the full month name (e.g. January). The *mm*, *dd* and *yy* terms

are 2-character month, day, and year, respectively, while *d* is the date with the leading zero suppressed. The *yyyy* term is the 4-character year obtained by adding 1980 to the first byte of the date array.

For all modes except 0, a null byte is appended to the output string.

RETURNS

The function does not make validity checks on the date array thus it cannot fail. It returns a pointer to the first byte past the generated output. For modes other than 0, this is a pointer to the null terminator.

SEE ALSO

stptime, getclk, getft

NAME

stpsym Get next symbol from a string

SYNOPSIS

```
#include <string.h>

p = stpsym(s,sym,symlen);

char *p; points to next input character
char *s; input string
char *sym; output string
int symlen; sizeof(sym)
```

DESCRIPTION

This function breaks out the next symbol from the input string. The first character of the symbol must be alphabetic (upper or lower case), and the remaining characters must be alphanumeric. Note that the pointer is not advanced past any initial white space in the input string.

The output string is the null-terminated symbol, and will be an empty string if no symbol is found. If the symbol is longer than **symlen-1**, its excess characters are dropped.

RETURNS

The function returns a pointer to the next character past the symbol.

SEE ALSO

stcarg, stpbrk, strcspn, strpbrk

EXAMPLES

```
#include <stdio.h>
#include <string.h>

main()
{
char a[256],b[10];

while(1)
  {
  printf("\nEnter text string:   ");
  if(gets(a) == NULL) exit(0);
  While(1)
    {
    p = stpsym(a,b,sizeof(b));
    printf("Symbol: \"%s\"  Residual: \"%s\"\n",b,p);
    if(b[0] == '\0') break;
    }
  }
}
```

NAME

stptime Convert time array to string

SYNOPSIS

```
#include <string.h>

np = stptime(p,mode,time);

char *np; updated output string pointer
char *p; output string pointer
int mode; conversion mode
char *time; time array, as follows
time[0] => hour (0 to 23)
time[1] => minute (0 to 59)
time[2] => second (0 to 59)
time[3] => hundredths (0 to 99)
```

DESCRIPTION

This function converts a 4-byte time array into ASCII or BCD according to the **mode** argument:

0 = >	hhmmssdd (BCD, 4 bytes)
1 = >	hhmmss (ASCII, 7 bytes)
2 = >	hh:mm:ss (ASCII, 9 bytes)
3 = >	hhmmssdd (ASCII, 9 bytes)
4 = >	hh:mm:ss.dd (ASCII, 12 bytes)
5 = >	hh:mm (ASCII, 6 bytes)
6 = >	hr:mm:ss HH (ASCII, 12 bytes)
7 = >	hr:mm HH (ASCII, 9 bytes)

The *hh*, *mm*, *ss* and *dd* terms are simply the 2-digit (BCD or ASCII) equivalents of the

binary values in the time array. The *hr* term is the 2-digit hour using the 12-hour form, and the *HH* term is either AM or PM.

Note that a null terminator is appended to the ASCII output strings.

RETURNS

The function does not make validity checks on the time array, thus it cannot fail. It returns a pointer to the first byte past the generated output. For modes other than 0, this is a pointer to the null terminator.

SEE ALSO

stpdate, getclk, getft

NAME

stptok Get next token from a string

SYNOPSIS

```
#include <string.h>

p = stptok(s,tok,toklen,brk);

char *p; points to next character after token
char *s; points to input string
char *tok; points to output buffer
int toklen; sizeof(tok)
char *brk; break string
```

DESCRIPTION

This function breaks out the next token from the input string and moves it to the token buffer with a null terminator. A token consists of all characters in the input string **s** up to but not including the first character that is in the break string. In other words, **brk** specifies the characters that cannot be included in a token.

If the input string begins with a break character, then the token buffer will contain a null string, and the return pointer **p** will be the same as **s**. If no break character is found after **toklen-1** input characters have been moved to the token buffer, or if the input string terminator (a null byte) is hit, then the scan stops as if a break character were hit.

RETURNS

The function returns a pointer to the next character in the input string.

NOTE: This function does not delete white space at the beginning of the input string.

SEE ALSO

stpblk, strtok

EXAMPLES

```
/*
*
* This example breaks out words that are
* separated by blanks or commas.
* The token buffer takes on the following
* values as the program loops:
*
* LOOP TOKEN
*  1 "first"
*  2 "second"
*  3 "third"
*  4 "fourth"
*
*/
#include <string.h>
#include <stdio.h>

char test[ ] = "first, second third, fourth";

main()
{
char *p = test;
char token[50];

while(1)
        {
        p = stptok(p,token,sizeof(token)," ,");
        printf("%s\n",token);
        if(*p == '\0') break;
        p = stpblk(++p);
        }
}
```

NAME

strbpl Build string pointer list

SYNOPSIS

```
#include <string.h>

n = strbpl(s,max,t);

int n; number of pointers
char *s[]; pointer to string pointer list
int max; maximum number of pointers
char *t; text pointer
```

DESCRIPTION

This function constructs a list of pointers to the strings contained within the specified text array. Each string must be null-terminated, and the text array must be terminated by a null string. In other words, array **t** must end with two null bytes, one to terminate the final string and another to terminate the array. The string pointer list **s** is terminated by a null pointer.

RETURNS

The return value indicates how many string pointers were placed into array **s**. If the number of strings plus the final null pointer is greater than **max**, a value of -1 is returned.

SEE ALSO

getfnl, strsrt

EXAMPLES

```
char text[ ] = string 1,"string 2",'\0';
char *list[5];
int count;

/*
* The following call has the following effect:
*
* Return value (count) is 2.
* list[0] => "string 1"
* list[1] => "string 2"
* list[2] => NULL
*
*/
count = strbpl(list,5,text);
```

NAME

strcat, strncat	Concatenate strings
strcat	Concatenate strings
strncat	Concatenate strings, max length

SYNOPSIS

```
#include <string.h>

p = strcat(to,from);
p = strncat(to,from,n);

char *p; same as destination string pointer
char *to; destination string pointer
char *from; source string pointer
int n; maximum source length
```

DESCRIPTION

These functions concatenate the source string to the tail end of the destination string. For **strncat**, no more than **n** characters are moved from the source to the destination. A null byte is placed at the end of the destination in any case.

RETURNS

Both functions return a pointer that is the same as the first argument.

SEE ALSO

stpcpy

EXAMPLES

```
#include <stdio.h>
#include <string.h>

main()
{
char a[256],b[256];
int n;

while(1)
    {
    printf("\nEnter string A: ");
    if(gets(a) == NULL) exit(0);
    printf("Enter string B: ");
    if(gets(b) == NULL) exit(0);
    printf("Enter maximum length N: ");
    scanf("%d",&n);
    printf("strcat(A,B):     \"%s\"\n",strcat(a,b));
    printf("strncat(A,B,N): \"%s\"\n",strncat(a,b,n));
    }
}
```

strcmp, stricmp, strncmp, strnicmp

Compare strings *Class: ANSI*

NAME

strcmp, stricmp, strncmp, strnicmp	Compare strings
strcmp	Compare strings
strcmpi	Compare strings, case-insensitive
stricmp	Compare strings, case-insensitive
strncmp	Compare strings, length-limited
strnicmp	Compare strings, no case, max size

SYNOPSIS

```
#include <string.h>

x = strcmp(a,b);
x = strcmpi(a,b);
x = stricmp(a,b);
x = strncmp(a,b,n);
x = strnicmp(a,b,n);

int x; comparison result
char *a,*b; strings being compared
```

DESCRIPTION

These functions compare two null-terminated strings. The ASCII collating sequence is used in all cases, but **strcmpi**, **stricmp** and **strnicmp** do not distinguish between upper and lower case. Note that **strcmpi** is exactly the same as **stricmp**, except that the former is a hold-over from various Microsoft compilers, while the latter is in the ANSI standard.

The relative collating sequence of the strings is indicated by the sign of the return value, as follows:

NEGATIVE = > first string is below second

ZERO = > strings are equal

POSITIVE = > first string is above second

If the strings have different lengths, the shorter one is treated as if it were extended with zeroes. For **strncmp** and **strnicmp**, no more than **n** characters are compared.

RETURNS

As noted above.

EXAMPLES

```
#include <stdio.h>
#include <string.h>

main()
{
char *p,a[256],b[256];
int n,x;

while(1)
  {
  printf("Enter string A: ");
  if(gets(a) == NULL) exit(0);
  printf("Enter string B: ");
  if(gets(b) == NULL) exit(0);
  printf("Enter maximum compare length: ");
  scanf("%d",&n);

  result("strcmp:   ",strcmp(a,b));
  result("stricmp: ",stricmp(a,b));
  result("strncmp: ",strncmp(a,b,n));
  result("strnicmp:",strnicmp(a,b,n));
  }
}
```

Compare strings *Class: ANSI*

```
void result(name,r)
char *name;
int r;
{
char *p;

if(x == 0) p = "is equal to";
if(x < 0) p = "is less than";
if(x > 0) p = "is greater than";
printf("%s String A %s string B\n",name,p);
}
```

NAME

strdup Duplicate a string

SYNOPSIS

```
#include <string.h>

p = strdup(s);

char *p; points to duplicate string
char *s; points to string being duplicated
```

DESCRIPTION

This function creates a duplicate of the specified string by using **malloc** and **strcpy** to allocate space and copy the string to it.

RETURNS

A null pointer is returned if **malloc** fails. Otherwise, the function returns a pointer to the duplicate string.

NAME

strins Insert a string

SYNOPSIS

```
#include <string.h>

void strins(to,from);

char *to; destination string
char *from; source string
```

DESCRIPTION

This function inserts the source string in front of the destination. Both strings must be null-terminated, and the destination is shifted to the right (upward in memory) in order to accomodate the source string. The final result is a single null-terminated string.

> NOTE: Ensure that the destination area is large enough, otherwise you could have a systems failure.

SEE ALSO

strcat

EXAMPLES

```
#include <string.h>
char here[] = "Here ";
char now[30] = "and now";
printf("%s, %s\n",here,now);
strins(now,here); /* now => "Here and now" */
printf("%s\n",now);
```

NAME

strlen,stclen	Measure length of a string
stclen	Measure length of a string
strlen	Measure length of a string

SYNOPSIS

```
length = strlen(s);
length = stclen(s);
int length; number of bytes in 's' (before null)
```

DESCRIPTION

These functions return the number of bytes in string **s** before the null terminator byte.

RETURNS

length = number of bytes in string before null byte

EXAMPLES

```
x = strlen("abc"); /* x is 3 */
x = strlen("");  /* x is 0 */
```

NAME

strlwr,strupr	Change case of string
strlwr	Convert string to lower case
strupr	Convert string to upper case

SYNOPSIS

```
#include <string.h>

p = strlwr(s);
p = strupr(s);

char *p; return pointer (same as s)
char *s; string pointer
```

DESCRIPTION

These functions convert all alphabetic characters in the specified null-terminated string to lower or upper case. In each case, the function return value is the same as the string pointer.

RETURNS

Both functions return the original string pointer.

NAME

strmfe Make file name iwht extension

SYNOPSIS

```
#include <string.h>

strmfe(newname,oldname,ext);

char *newname; new file name
char *oldname; old file name
char *ext; extension
```

DESCRIPTION

This function copies the old file name to the new name, deleting any extension. Then it appends the specified extension to the new file name, with an intervening period. For example:

OLDNAME	EXT	NEWNAME
c:myprog.c	*cc*	*c:myprog.cc*
abc	*exe*	*abc.exe*

NOTE: The **newname** area must be large enough to accept the file name string and the separator. A safe size is **FMSIZE**, which is defined in the **dos.h** header file.

SEE ALSO

strmfn, strmfp

NAME

strmfn Make file name from components

SYNOPSIS

```
#include <string.h>

strmfn(file,drive,path,node,ext);

char *file; file name pointer
char *drive; drive code pointer
char *path; directory path pointer
char *node; node pointer
char *ext; extension pointer
```

DESCRIPTION

This function makes a file name from four possible components. In general, the name is constructed as follows:

```
drive:path/node.ext
```

If the **drive** pointer is not null, that string is moved to the area pointed to by the **file** argument. Then a colon is inserted unless one is already there. Next, if **path** is not NULL, it is appended to **file**, and the directory separator specified by **_SLASH** is added if necessary. The **node** string is appended next, unless it is NULL. Finally, if **ext** is not NULL, a period is appended to **file**, followed by the **ext** string.

RETURNS

None

> NOTE: Ensure the **file** pointer refers to an area which is large enough to hold the result. A safe value is **FMSIZE**, which is defined in **dos.h**.

SEE ALSO

strmfe, strmfp, _SLASH

EXAMPLES

```
#include <dos.h>
#include <stdio.h>
#include <string.h>

char buffer[FMSIZE];

/* The next statements both place "abc/def/ghi"
   into the buffer.   */


strmfn(buffer,NULL,"abc/def","ghi",NULL);
strmfn(buffer,NULL,"abc/def/","ghi",NULL);


/* The next statements both generate
   "df0:myfile.str"  */

strmfn(buffer,"df0",NULL,"myfile","str");
strmfn(buffer,"df0:",NULL,"myfile","str");
```

NAME

strmfp Make file name from path/node

SYNOPSIS

```
#include <string.h>

strmfp(name,path,node);

char *name; file name
char *path; directory path
char *node; node
```

DESCRIPTION

This function copies the path string to the file name area, appending the **_SLASH** separator if the path string is not empty and does not end with a slash, backslash, or colon. Then the node string is appended to the file name. **_SLASH** is an external character variable that defaults to a slash (/).

> NOTE: The **name** area must be large enough to accept the file name string. A safe value is **FMSIZE**, which is defined in the *dos.h* header file.

SEE ALSO

strmfe, strmfn, _SLASH

NAME

strnset, strset	Set string to length
strnset	Set string to value, max length
strset	Set string to value

SYNOPSIS

```
#include <string.h>

p = strnset(s,c,n);
p = strset(s,c);

char *p; return pointer (same as s)
char *s; string pointer
int c; value
int n; maximum string length
```

DESCRIPTION

These functions set all bytes of a null-terminated string to the same value, not including the terminator byte. For **strnset** no more than **n** bytes will be set. That is, if a null byte is not found by the time **n** bytes have been set, the operation stops.

RETURNS

Both functions return the original string pointer.

NAME

strrev Reverse a character string

SYNOPSIS

```
#include <string.h>

p = strrev(s);

char *p,*s; string pointer
```

DESCRIPTION

This function reverses a character string. In other words it *rotates* the string about its mid-point such that the last character is first and the first is last.

RETURNS

Returns same pointer that was passed to it.

NAME

strsfn Split the file name

SYNOPSIS

```
#include <string.h>

strsfn(file,drive,path,node,ext);

char *file; file name pointer
char *drive; drive code pointer
char *path; directory path pointer
char *node; node pointer
char *ext; extension pointer
```

DESCRIPTION

This function splits a file name into four possible components and places them into the **drive**, **path**, **node** and **ext** strings. If any of those arguments are NULL, then those components are discarded.

In general, a complete file name is constructed as follows:

```
drive:path/node.ext
```

When **strsfn** splits the file name, it leaves the colon attached to the drive code, but drops the other punctuation characters. In other words, the **path** component will not end with a slash, and the **ext** component will not begin with a period. Slashes or backslashes within the **path** component are preserved.

RETURNS

None

SEE ALSO

strgfn, strmfe, strmfn

EXAMPLES

```
#include <dos.h>
#include <stdio.h>
#include <stdlib.h>

char a[3],b[FMSIZE],c[FNSIZE],d[4];

/* After the next statement, the component strings are:

        a => ""
        b => "abc/def"
        c => "ghi"
        d => ""

*/

strsfn("abc/def/ghi",a,b,c,d);

/* After the next statement, the component strings are:

        a => "b:"
        b => ""
        c => "myfile"
        d => "str"

*/

strsfn("b:myfile.str",a,b,c,d);
```

NOTE: You must make sure that the **drive**, **path**, **node** and **ext** pointer refer to areas that are large enough to hold the largest string that might be generated. Note that **strsfn** does not check that any component lengths are exceeded, although it does copy **file** string to an internal buffer of size FMSIZE and truncate it if it is too long. If you want to be absolutely sure that no overflows occur, make each component area be FMSIZE bytes long.

EXAMPLES

```
/*
*
* This program splits file names that it reads
* from stdin.
*
*/
#include <stdio.h>
#include <string.h>

main()
{
char input[80],drive[80],path[80],node[80],ext[80];

while(1)
        {
        puts("\nEnter file name...\n");
        if(gets(input) == NULL) exit(0);
        strsfn(input,drive,path,node,ext);
        printf("DRIVE: \"%s\"\n",drive);
        printf("PATH:  \"%s\"\n",path);
        printf("NODE:  \"%s\"\n",node);
        printf("EXT:   \"%s\"\n",ext);
        }
}
```

NAME

strsrt Sort string pointer list

SYNOPSIS

```
#include <string.h>

strsrt(s,n);

char *s[]; string pointer list
int n; number of pointers in list
```

DESCRIPTION

This function performs a simple bubble sort of the string pointers in the specified list. It is particularly useful in conjunction with the **getfnl** and **strbpl** functions. For large lists, you will usually get better performance using **tqsort**.

RETURNS

None.

SEE ALSO

getfnl, strbpl, tqsort

EXAMPLES

```
/*
* This program constructs an array of pointers
* to all file names in the current directory
* that have an extension of ".c".  Then the
* array is sorted into ASCII order.
*
*/
```

```
char names[3000],*pointers[300];
int count;

count = getfnl("*.c",names,sizeof(names),0);
if(count > 0)
   {
   if(strbpl(pointers,300,names) != count) goto error;
   strsrt(pointers,count);
   }
```

NAME

strtok Get a token

SYNOPSIS

```
#include <string.h>

t = strtok(s,b);

char *t; token pointer
char *s; input string pointer or NULL
char *b; break character string pointer
```

DESCRIPTION

This function treats the input string as a series of one or more tokens separated by one or more characters from the break string. By making a sequence of calls to **strtok**, you can obtain the tokens in left-to-right order. To get the first (leftmost) token, supply a non-NULL pointer for the **s** argument. Then to get the next tokens, call the function repeatedly with a NULL pointer for **s**, until you get a NULL return pointer to indicate that there are no more tokens. The break string can be changed from one call to another.

Each time it is entered, **strtok** takes the following steps:

1. If the input string is NULL, obtain the string pointer that was used on the preceding call. Otherwise use the new input string pointer.

2. Scan forward through the string to the next non-break character. If it is a null byte, return a value of NULL to indicate that there are no more tokens.

3. Scan forward through the string to the next break character or the null terminator. In the former case, write a null byte into the string to terminate the token, and then scan forward until the next non-break is found. In either case, save the final value of the string pointer for the next call, and return the token pointer.

Note that the input string gets changed as the scan progresses. Specifically, a null byte is written at the end of each token.

RETURNS

A NULL pointer is returned when there are no more tokens.

SEE ALSO

stptok, strcspn, strspn

EXAMPLES

```
/*
* This example breaks out words that are separated
* by blanks or commas. The token pointer takes on
* the following values as the program loops:
*
* LOOP  TOKEN
*   1 "first"
*   2 "second"
*   3 "third"
*   4 "fourth"
*   5 NULL
*
*/
#include <string.h>
#include <stdio.h>

char test[ ] = "first, second third, fourth";

char *token;
token = strtok(test,", ");
while(token != NULL)
   {
   printf("%s\n",token);
   token = strtok(NULL,", ");
   }
}
```

NAME

strtol Convert string to long integer

SYNOPSIS

```
#include <string.h>

r = strtol(p,np,base);

long r; result
char *p; input string pointer
char **np; receives new input string pointer
int base; conversion base
```

DESCRIPTION

This function converts an ASCII input string into a long integer according to the specified base, which can range from 0 to 36, excluding 1. Valid digit characters are 0 to 9, a to z, and A to Z. The highest allowable character is determined by the conversion base. For example, if the base is 17, then the string can contain digits from 0 to 9, a to g, and A to G.

The function skips leading white space and then checks for a leading plus or minus sign. In the latter case, the result of the conversion is negated before it is returned. The conversion stops at the first invalid character, and a pointer to that character is returned in **np** if the **np** argument is not NULL. Note that if the entire string is converted, **np** will contain a pointer to the null terminator byte.

If **base** is 0, then the string is analyzed to see if it is octal, decimal, or hexadecimal, as follows:

Base 16 If the string begins with 0x or 0X, base 16 (hexadecimal) conversion is performed.

Base 8	Otherwise, if the string begins with 0, base 8 (octal) conversion is performed.
Base 10	If neither of the above applies, base 10 (decimal) conversion is performed.

> NOTE: There is no check is made for overflow.

SEE ALSO

atol, stcd_l

EXAMPLES

```
/*
*
* This program tests the strtol function.
*
*/
#include <stdio.h>
#include <string.h>

main()
{
char *p,buff[80];
int base;
long x;

while(1)
        {
        printf("\nEnter number base (0 to 36): ");
        if(gets(buff) == NULL) exit(0);
        if(buff[0] == '\0') exit(0);
        base = atoi(buff);
        if((base < 0) || (base > 36)) continue;

        printf("Enter number: ");
        if(gets(buff) == NULL) exit(0);
        if(buff[0] == '\0') exit(0);
        x = strtol(buff,&p,base);
        printf("Decimal result = %ld\n",x);
```

```
        if(*p != '\0') printf("Residual = %s\n",p);
        }
}
```

NAME

stspfp Parse file path

SYNOPSIS

```
#include <string.h>

error = stspfp(path,nx);

int error; -1 for error, 0 for success
char *path; file path string
int nx[16]; node index array
```

DESCRIPTION

This function parses a file path, which is a null-terminated string consisting of nodes separated by the **_SLASH** character. Each separator is replaced with a null byte, and the index to the first character of that node is placed into the node index array. The last entry in the array is followed by a -1. A leading separator in the path string is skipped.

RETURNS

A return value of -1 indicates that the path contains more than 15 nodes.

SEE ALSO

stcgfe, stcgfn, stcgfp, strsfn

EXAMPLES

```
/*
*
* The following piece of code parses /ABC/DE/F
* into strings ABC, DE, and F. The node index
* array will then contain 1, 5, 8, and -1.
*
*/
int xx[16];

stspfp("/ABC/DE/F",xx);
```

NAME

system Call system command processor

SYNOPSIS

```
#include <stdlib.h>

error = system(cmd);

int error; non-zero if error
char *cmd; command string
```

DESCRIPTION

This function invokes the system command processor and passes the **cmd** string to it. Under AmigaDOS, this function invokes the AmigaDOS **Execute** facility.

RETURNS

If the command processor cannot be invoked, a value of -1 is returned, and additional error information can be found in **errno** and **_OSERR**. Otherwise, the function returns the value passed back by the command processor.

SEE ALSO

errno, _OSERR

EXAMPLES

```
#include <stdlib.h>
int x;
x = system("copy #? to df1:");
if(x < 0) printf("System command failed\n");
if(x > 0) printf("System command error %d\n",x);
```

time

Get system time in seconds *Class: ANSI*

NAME

time Get system time in seconds

SYNOPSIS

```
#include <time.h>

timeval = time(timeptr);

long timeval; time value
long *timeptr; pointer to time value storage
```

DESCRIPTION

This function returns the current time expressed as the number of seconds since 00:00:00 Greenwich Mean Time, January 1, 1970. If **timeptr** is not NULL, the time value is also stored in that location.

SEE ALSO

asctime, ctime, ftime, gmtime, localtime, tzset, utime

EXAMPLES

```
#include <time.h>
#include <stdio.h>

main()
{
long t;

time(&t);
printf("Current time is %s\n",ctime(&t));
}
```

NAME

to	Character conversion functions
toascii	Convert character to ASCII
tolower	Convert character to lower case
toupper	Convert character to upper case

SYNOPSIS

```
#include <ctype.h>

cc = toascii(c);
cc = tolower(c);
cc = toupper(c);

int cc; converted character
int c; character to convert
```

DESCRIPTION

These functions convert characters into different forms. If you include **ctype.h** as shown above, they are actually defined as macros and produce in-line code to perform the conversions. Without **ctype.h**, they are actual functions resolved in the standard library. If you want to use the function version but must include **ctype.h** for some other reason, use **#undef** to undefine the conversion macros.

The **toascii** conversion simply resets all high-order bits, leaving only the lower seven. The **tolower** conversion tests if **c** is an upper case alphabetic character and, if so, converts it to lower case. Otherwise, **cc** is the same as **c**. The **toupper** conversion is the reverse of **tolower**.

SEE ALSO

_ctype

EXAMPLES

```
/*
*
* The following program echoes each input
* line in upper case.
*
*/
#include <stdio.h>
#include <ctype.h>

main()
{
char b[100],*p;

while(gets(b) != NULL)
        {
        for(p = b; *p != '\0'; p++) *p = toupper(*p);
        puts(b);
        }
}
```

NAME

trig	Trigonometric functions
cos	Cosine function
sin	Sine function
tan	Tangent function
acos	Arccosine function
asin	Arcsine function
atan	Arctangent function
atan2	Arctangent of x/y
cosh	Hyperbolic cosine function
sinh	Hyperbolic sine function
tanh	Hyperbolic tangent function

SYNOPSIS

```
#include <math.h>

r = cos(x);
r = sin(x);
r = tan(x);
```

```
r = acos(x);
r = asin(x);
r = atan(x);
r = atan2(x,y);

r = cosh(x);
r = sinh(x);
r = tanh(x);

double r; result;
double x,y; arguments
```

DESCRIPTION

The **cos**, **sin** and **tan** routines compute the normal circular functions of angles expressed in radians.

The **acos**, **asin**, **atan** and **atan2** routines compute the inverse circular functions, returning angular values expressed in radians. Results are constrained as follows:

FUNCTION	RETURN RANGE
acos	0 to PI
asin	-PI/2 to PI/2
atan	-PI/2 to PI/2
atan2	-PI/2 to PI/2

Since the tangent becomes very large for angles close to PI/2, the **atan2** function is often used to avoid computations with large numbers that might easily overflow. With **atan2**, you can express the large tangent value as a quotient of two more reasonable numbers.

The **cosh**, **sinh** and **tanh** routines compute the normal hyperbolic functions.

SEE ALSO

matherr

NAME

tzset Set time zone variable

SYNOPSIS

```
#include <time.h>

tzset();

/* These symbols are defined in time.h:
*
* extern int daylight;
* extern long timezone;
* extern char *tzname[2];
* extern char tzstn[4];
* extern char tzdtn[4];
*
*/
```

DESCRIPTION

The **tzset** function assigns values to the time zone variables **daylight**, **timezone** and **tzname**. These variables are then used by **localtime** and other functions to correct from Greenwich Mean Time (GMT) to local time.

The values for these variables are obtained from the character string pointer named **_TZ**, which has the form:

```
char *_TZ = "aaabbbccc"
```

where **aaa** is the 3-letter abbreviation for the local standard time zone (e.g. CST), and **bbb** is a number from -23 to +24 indicating the value that is subtracted from GMT in order to obtain local standard time. Both **aaa** and **bbb** are required, but **ccc** is the abbreviation for the local daylight savings time zone (e.g. CDT), and it should be present only if daylight savings time is currently in effect. Initially, **_TZ** is set to NULL.

It should be initialized with the address of a string corresponding to the correct time zone. If **_TZ** is NULL, **tzset** uses the default string *CDT6*.

When **tzset** is called, first **timezone** is loaded with the number of seconds that must be subtracted from GMT in orderd to get the local time. Next **daylight** is loaded with 0 if the **ccc** portion of **_TZ** is absent and 1 if **ccc** is present. Then the **aaa** and **ccc** parts are copied to **tzstn** and **tzdtn**, respectively, with null terminators. Finally, **tzname[0]** and **tzname[1]** are loaded with pointers to **tzstn** and **tzdtn**, respectively.

RETURNS

None.

> NOTE: The **tzstn** and **tzdtn** variables are Lattice extensions and will not exist on the typical UNIX system.

SEE ALSO

timedata, localtime

NAME

ungetc Push input character back

SYNOPSIS

```
r = ungetc(c, fp);
int r; return character or code
char c; character to be pushed back
FILE *fp; file pointer
```

DESCRIPTION

This function pushes a character back to the specified Level 2 input file. The character need not be the same as the one that was most recently read. However, before calling **ungetc**, you must have read at least one character via **fgetc** or one of the other Level 2 input functions. Also, you can only push back one character; if you call **ungetc** more than once between input functions, the results are undefined.

RETURNS

Normally **ungetc** returns the character that was pushed back. However, if the end-of-file has been hit or if no characters have been read yet, the value EOF is returned.

EXAMPLES

```
#include <stdio.h>

main()
{
int c;

while(1)
  {
  printf("Loop 1...\n");
  while((c = getchar()) != EOF)
    {
```

```
        if(isalpha(c)) putchar(c);
        else break;
        }
    ungetc(c);
    printf("\n\nLoop 2...\n");
    while((c = getchar()) != EOF)
        {
        if(isalpha(c) == 0) putchar(c);
        else break;
        }
    ungetc(c);
    }
printf("\n\nDone\n");
}
```

NAME

utpack, utunpk	Pack or unpack UNIX time
utpack	Pack UNIX time
utunpk	Unpack UNIX time

SYNOPSIS

```
#include <stdlib.h>

ut = utpack(x);
utunpk(ut,x);

long ut; packed UNIX time
char *x; unpacked UNIX time
```

DESCRIPTION

These functions pack and unpack the 32-bit value time that is traditionally used in UNIX systems. This value is the number of seconds since 00:00:00, January 1, 1970. The **time** function returns the system clock in this form relative to Greenwich Mean Time.

The unpacked time is a 6-byte array in the following format:

x[0] = >	year - 1970 (-128 to +127)
x[1] = >	month (1 to 12)
x[2] = >	day (1 to 31)
x[3] = >	hour (0 to 23)
x[4] = >	minute (0 to 59)
x[5] = >	second (0 to 59)

Although this array is similar to the one produced by **getclk** and used by **stpdate**, note that the year is biased relative to 1970 instead of 1980. So, if you use **utunpk** followed by **stpdate**, you must subtract 10 from **x[0]** before the **stpdate** call. Note also that the year is a signed character and can be negative. A value of -3, for example, is 1967 (i.e. 1970 - 3).

SEE ALSO

ctime, getclk, gmtime, localtime, stpdate, time

EXAMPLES

```
/*
*
* Get a file time and convert it to UNIX time.
* No error checks.
*
*/
#include <time.h>
#include <dos.h>
#include <stdlib.h>
main(argc,argv)
int argc;
char *argv[ ];
{
char tt[6];
int fh;
long ft,ut;

ft = getft(argv[1]);
tt[0] -= 10;
ut = utpack(tt);
printf("File time is: %s\n",ctime(&ut));
}
```

Appendix G. Combined Reference

Name	Purpose	Page	Type
_bufsize	Level 2 I/O buffer size	D-13	LATTICE
_exit	Terminate program execution	F-54	ANSI
_fmode	Default level 2 I/O mode	D-14	LATTICE
_FPERR	Floating Point Error Code	D-15	LATTICE
_MNEED	Minimum Dynamic Memory Needed	D-17	LATTICE
_mstep	Memory Pool Increment Size	D-18	LATTICE
_OSERR	DOS Error Information	D-19	AMIGA
_SLASH	Directory separator character	D-22	LATTICE
_timedata	Time Zone variables	D-12	UNIX
abort	Abort the current process	F-2	ANSI
abs	Absolute value	F-3	ANSI
access	Check file accessibility	F-5	UNIX
argopt	Get options from argument list	F-7	LATTICE
asctime	Generate ASCII time string	F-10	AMIGA
asm	68000 Macro Assembler	C-2	LATTICE
assert	Assert program validity	F-12	ANSI
assign	Logical name assignment	C-6	AMIGA
atof	Convert ASCII to float	F-14	ANSI
atoi	Convert ASCII to integer	F-16	UNIX
atoil	Convert ASCII to integer	F-16	UNIX
calloc	Level 3 memory allocation	F-18	ANSI
ceil	Get floating point limits	F-22	ANSI
chdir	Change current directory	F-23	UNIX
chgclk	Get or change system clock	F-110	AMIGA
chkabort	Check for Break character	F-24	AMIGA
chkml	Check for largest memory block	F-25	LATTICE

Name	Purpose	Page	Type
chkufb	Check Level 1 file handle	F-26	LATTICE
chmod	Change file protection mode	F-27	UNIX
clearerr	Clear Level 2 I/O error flag	F-29	ANSI
close	Close a Level 1 file	F-31	UNIX
clrerr	Clear Level 2 I/O error flag	F-29	ANSI
creat	Create a Level 1 file	F-33	UNIX
ctime	Convert time value to string	F-35	ANSI
CXFERR	Low-level float error	F-37	LATTICE
DiskfontBase	Disk Font Library Vector	D-6	AMIGA
dclose	Close an AmigaDOS file	F-39	AMIGA
dcreat	Create AmigaDOS file	F-40	AMIGA
dcreatx	Create AmigaDOS file	F-40	AMIGA
dfind	Find directory entry	F-42	AMIGA
dnext	Find directory entry	F-42	AMIGA
dopen	Open an AmigaDOS file	F-44	AMIGA
DOSBase	AmigaDOS Library Vector	D-2	AMIGA
drand	Generate random numbers	F-45	UNIX
dread	Read and write AmigaDOS files	F-48	AMIGA
dseek	Re-position an AmigaDOS file	F-50	AMIGA
dwrite	Read and write AmigaDOS files	F-48	AMIGA
ecvt	Convert float to string	F-52	UNIX
errno	UNIX error number	D-3	UNIX
exit	Terminate program execution	F-54	ANSI
exp	Exponential functions	F-56	ANSI
fclose	Close a Level 2 file	F-58	ANSI
fcloseall	Close a Level 2 file	F-58	ANSI

Name	Purpose	Page	Type
fcvt	Convert float to string	F-52	UNIX
fdopen	Attach Level 1 file to Level 2	F-60	UNIX
feof	Check Level 2 file status	F-61	ANSI
ferror	Check Level 2 file status	F-61	ANSI
fflush	Flush Level 2 output buffer	F-62	ANSI
fgetc	Get a character	F-64	ANSI
fgets	Get string from a Level 2 file	F-66	ANSI
fileno	Get file number for a Level 2 file	F-69	UNIX
floor	Get floating point limits	F-22	ANSI
flushall	Flush Level 2 output buffer	F-62	ANSI
fmod	Float modulus operations	F-70	ANSI
fmode	Change mode of Level 2 file	F-72	LATTICE
fopen	Open a Level 2 file	F-73	ANSI
fork	Create a child process	F-76	LATTICE
fprintf	Formatted print	F-82	ANSI
fputc	Put a character	F-89	ANSI
fputchar	Put a character	F-89	ANSI
fputs	Put string to Level 2 file	F-91	ANSI
fread	Read and write blocks	F-93	ANSI
free	Level 3 memory allocation	F-18	ANSI
freopen	Reopen a Level 2 file	F-95	ANSI
frexp	Split or combine float value	F-97	ANSI
fscanf	Formatted input conversions	F-99	ANSI
fseek	Set or get Level 2 file position	F-104	ANSI
ftell	Set or get Level 2 file position	F-104	ANSI
fwrite	Read and write blocks	F-93	ANSI

Name	Purpose	Page	Type
gcvt	Convert float to string	F-106	UNIX
getc	Get a character	F-64	ANSI
getcd	Get current directory	F-108	AMIGA
getchar	Get a character	F-64	ANSI
getclk	Get or change system clock	F-110	AMIGA
getcwd	Get current working directory	F-112	UNIX
getdfs	Get free disk space	F-114	AMIGA
getfa	Get file attribute	F-116	AMIGA
getfnl	Get filename list	F-117	LATTICE
getft	Get file time	F-120	AMIGA
getmem	Get Level 2 memory block	F-121	LATTICE
getml	Get Level 2 memory block	F-121	LATTICE
gets	Get string from a Level 2 file	F-66	ANSI
GfxBase	Graphics Library Vector	D-7	AMIGA
gmtime	Unpack Greenwich Mean Time (GMT)	F-124	ANSI
include:	#include search path	E-2	AMIGA
IntuitionBase	Intuition Library Vector	D-8	AMIGA
iomode	Change mode of Level 1 file	F-126	LATTICE
is	Character tests	F-128	ANSI
lc	Lattice C Compiler	C-8	LATTICE
lc1	Compiler pass 1	C-19	LATTICE
lc2	Compiler pass 2	C-21	LATTICE
lc:	Compiler executable file path	E-3	AMIGA
ldexp	Split or combine float value	F-97	ANSI
lib:	Library file path	E-4	AMIGA
longjmp	Perform long jump	F-131	ANSI

Name	Purpose	Page	Type
lsbrk	Level 1 memory allocation	F-133	UNIX
lseek	Set or get file position	F-135	UNIX
main	Your main or principle program	F-138	ANSI
malloc	Level 3 memory allocation	F-18	ANSI
MathBase	FFP Library Vector	D-9	AMIGA
matherr	Math error handler	F-141	UNIX
MathTranBase	FFP TRig Library Vector	D-11	AMIGA
memory	Memory block operations	F-144	UNIX
mkdir	Create a new directory	F-147	UNIX
modf	Float modulus operations	F-70	ANSI
msflag	MS-DOS File Pattern Flag	D-10	LATTICE
nofill	Open a Level 1 file	F-152	UNIX
omd	Object Module Disassembler	C-23	LATTICE
oml	Object Module Librarian	C-25	LATTICE
onbreak	Plant break trap	F-148	AMIGA
onexit	Exit trap	F-150	ANSI
perror	Print UNIX error message	F-154	ANSI
poserr	Print AmigaDOS error message	F-156	AMIGA
putc	Put a character	F-89	ANSI
putchar	Put a character	F-89	ANSI
puts	Put string to Level 2 file	F-91	ANSI
qsort	Sort a data array	F-158	UNIX
quad:	Intermediate file path	E-5	AMIGA
rand	Simple random number generator	F-160	ANSI
rbrk	Level 1 memory allocation	F-133	UNIX
read	Read or write a Level 1 file	F-162	UNIX

Name	Purpose	Page	Type
realloc	Level 3 memory allocation	F-18	ANSI
remove	Remove a file	F-164	ANSI
rename	Rename a file	F-166	ANSI
rewind	Set or get Level 2 file position	F-104	ANSI
rlsmem	Release a Level 2 memory block	F-168	LATTICE
rlsml	Release a Level 2 memory block	F-168	LATTICE
rmdir	Remove a directory	F-171	UNIX
sbrk	Level 1 memory allocation	F-133	UNIX
setbuf	Set buffer mode for a Level 2 file	F-172	ANSI
setjmp	Perform long jump	F-131	ANSI
signal	Establish event traps	F-175	ANSI
sizmem	Get Level 2 memory pool size	F-177	LATTICE
srand	Simple random number generator	F-160	ANSI
stcarg	Get an argument	F-178	LATTICE
stccpy	Copy one string to another	F-180	ANSI
stcd_i	Convert strings to integer	F-182	LATTICE
stcgfe	Get file name components	F-184	LATTICE
stcgfn	Get file name components	F-184	LATTICE
stcgfp	Get file name components	F-184	LATTICE
stci_d	Convert integers to strings	F-188	LATTICE
stcis	Measure character set span	F-186	ANSI
stcisn	Measure character set span	F-186	ANSI
stclen	Measure length of a string	F-217	UNIX
stcpm	Pattern match functions	F-191	LATTICE
stcpma	Pattern match functions	F-191	LATTICE
stpblk	Skip blanks (white space)	F-194	ANSI

Name	Purpose	Page	Type
stpbrk	Find break character in string	F-196	ANSI
stpchr	Find character in string	F-198	ANSI
stpchrn	Find character in string	F-198	ANSI
stpcpy	Copy one string to another	F-180	ANSI
stpdate	Convert date array to string	F-200	LATTICE
stpsym	Get next symbol from a string	F-202	LATTICE
stptime	Convert time array to string	F-204	LATTICE
stptok	Get next token from a string	F-206	LATTICE
strbpl	Build string pointer list	F-208	LATTICE
strcat	Concatenate strings	F-210	ANSI
strchr	Find character in string	F-198	ANSI
strcmp	Compare strings	F-212	ANSI
strcpy	Copy one string to another	F-180	ANSI
strcspn	Measure character set span	F-186	ANSI
strdup	Duplicate a string	F-215	XENIX
stricmp	Compare strings	F-212	ANSI
strins	Insert a string	F-216	LATTICE
strlen	Measure length of a string	F-217	UNIX
strlwr	Change case of string	F-218	XENIX
strmfe	Make file name iwht extension	F-219	LATTICE
strmfn	Make file name from components	F-221	LATTICE
strmfp	Make file name from path/node	F-223	LATTICE
strncat	Concatenate strings	F-210	ANSI
strncmp	Compare strings	F-212	ANSI
strncpy	Copy one string to another	F-180	ANSI
strnicmp	Compare strings	F-212	ANSI

Name	Purpose	Page	Type
strnset	Set string to length	F-224	XENIX
strpbrk	Find break character in string	F-196	ANSI
strrchr	Find character in string	F-198	ANSI
strrev	Reverse a character string	F-225	XENIX
strset	Set string to length	F-224	XENIX
strsfn	Split the file name	F-226	LATTICE
strspn	Measure character set span	F-186	ANSI
strsrt	Sort string pointer list	F-229	LATTICE
strtok	Get a token	F-231	ANSI
strtol	Convert string to long integer	F-233	UNIX
strupr	Change case of string	F-218	XENIX
stspfp	Parse file path	F-236	LATTICE
system	Call system command processor	F-238	ANSI
tell	Set or get file position	F-135	UNIX
time	Get system time in seconds	F-240	ANSI
to	Character conversion functions	F-241	ANSI
trig	Trigonometric functions	F-243	UNIX
tzset	Set time zone variable	F-246	XENIX
ungetc	Push input character back	F-248	ANSI
unlink	Remove a file	F-164	ANSI
utpack	Pack or unpack UNIX time	F-250	LATTICE
utunpk	Pack or unpack UNIX time	F-250	LATTICE
write	Read or write a Level 1 file	F-162	UNIX

Appendix I. Index

6502 Processor 2-1
68000 Operations 5-6
68010 Processor 2-2, 5-4
68020 Processor 2-2, 5-4
8-bit Processors 2-1
8086 Processor 2-5, 5-4
8088 Processor 5-4

A

A1000 1-1
A2000 Model 2-2
A2000 1-1
A500 Model 2-2
A500 1-1
abs F-3
access F-5
Accessible Libraries 2-5
Address Alignment 5-4
Address Registers 3-8
Addressing Modes 5-10
AddTask() Function 5-11, 8-7
Aggregate Assignment 4-20
Aggregate Passing Values 4-20
Alerts 2-1
ALINK 6-1, 8-1, 8-7
Amiga Operating System 2-1
amiga.lib 3-6
AmigaDOS ED 3-1
AmigaDOS EDIT 3-1
ANSI Compatibility 3-11, 4-1, 4-9, 4-11
ANSI Standard 3-10
ANSI 9-3
Apple MACINTOSH 3-13
argopt F-7
Argument Expression Errors 4-21
Argument Type Checking 4-1, 9-6

Argument Types 4-21
argv Function 9-2
argv Pointers 8-9
Arithmetic Object Initialization 4-15
Arithmetic Objects 4-10
ASCII Files 3-1, 3-12
asctime F-10
ASM C-2, 8-12
Assembly Language Interfacing 5-6
Assembly Language Modules 5-8
assert F-12
ASSIGN Command 1-3, 1-6
assign C-6
Assignment Operators 4-5
atof F-14
atoi F-16
atoil F-16
Automatic Link Vector 6-12
Automatic Load Vector 6-19
Automatic Objects 4-12

B

_BackGroundIO Flag 5-13
Base Relative Addressing 5-10
Beginners Errors 3-15
BLINK Command Line Examples 6-6
BLINK Keywords 6-1
BLINK 5-12, 8-12
Blitter 2-1, 8-3
Break Characters 8-2
_bufsize D-13
Built-In Function Advantages 3-10
Built-In Functions 3-10
Bus Contention 8-3

C

calloc F-18
CBack.o 5-12
ceil F-22
Character Constants 4-3
chdir F-23
chgclk F-110
CHIP Memory 8-3
chkabort F-24
chkml F-25
chkufb F-26
chmod F-27
Circular Definitions 4-9
clearerr F-29
clist.library 2-3
close F-31
clrerr F-29
Code Massaging 5-6
Code Portability 2-5, 4-11
Commands 9-1
Compile-Time Limits 8-3
Compiler Advanced Features 3-15
Compiler Changes 8-2
Compiler Limitations 4-18
Complex Data Types 5-4
Conditional Compilation 4-8
Console Handling 8-9
Copper 2-1, 8-3
creat F-33
CSET Statement 5-8
ctime F-35
CTRL- Character 3-3
CTRL-C Handling 8-9
Current Directory 3-2
Current Location Counter 5-12
Custom Environment 1-6
Custom Overlays 6-11
CXFERR F-37

D

Data File Portability 5-6
Data Names 9-3
Data Portability 5-4
Data Representations 5-3
Data Type Conversions 4-4
Data Type Sizes 5-3
Data Types 3-4, 4-10
dclose F-39
dcreat F-40
dcreatx F-40
Debug Flag 8-6
Demonstration Programs 3-4
Derived Objects 4-12
dfind F-42
Direct Memory Channels 2-3
Directory Assignment 1-5
Directory Paths 9-6
Disk Drives 1-1
Disk-Based Libraries 2-5
diskfont.library 2-3
DiskfontBase D-6
dnext F-42
DoIO() Function 3-8
dopen F-44
dos.library 2-3
DOSBase D-2
drand F-45
dread F-48
dseek F-50
Duplicating Disks 1-3
dwrite F-48

E

ecvt F-52
Enum Keyword 4-2, 4-19
Environment Variables 9-7
Equality Operators 4-5

errno D-3
EXEC Lists 2-2
EXEC System Routines 2-1
EXEC 2-1
exec.library 2-2, 2-3, 3-9, 3-12
exit F-54
exp F-56
Expression Evaluation Order Assumptions 4-17
Expression Evaluation Order 4-15
External Data Definitions 4-8
External Functions 4-21
External Objects 4-12

F

FAST Memory 8-3
fclose F-58
fcloseall F-58
fcvt F-52
FD2PRAGMA 3-12
fdopen F-60
fd Files 3-9, 3-12
fd File Conversion 3-13
feof Function 8-1
feof F-61
ferror Function 8-1
ferror F-61
fflush F-62
fgetc F-64
fgets F-66
File Assignment 1-5
File Extensions 3-1
File Names 9-6
fileno Function 8-1
fileno F-69
Floating Point Differences 3-5
Floating Point 8-1
floor F-22
Floppy Disk Installation 1-3
flushall F-62

fmod F-70
_fmode D-14, F-72
fopen F-73
fork F-76
Formal Objects 4-12
_FPERR D-15
fprintf F-82
fputc F-89
fputchar F-89
fputs F-91
fread F-93
Free Memory Pool 5-2
free F-18
freopen F-95
frexp F-97
fscanf F-99
fseek F-104
ftell F-104
Function Arguments 8-1
Function Calling Sequences 9-4
Function Consistency 8-9
Function Entry Conditions 5-7
Function Names 9-3
Function Return Values 5-8
Functions Returning Aggregates 4-21
fwrite F-93

G

gcvt F-106
getc F-64
getcd F-108
getchar F-64
getclk F-110
getcwd F-112
getdfs F-114
getfa F-116
getfnl F-117
getft F-120
getmem F-121

getml F-121
gets F-66
GfxBase D-7
Global Variable _Backstdout 5-13
gmtime F-124
graphics.library 2-3

H

Hard Disks 1-2, 1-5
Hardware 1-1
Header File Compression 3-11
Header File Organization 3-11
Header Files 2-3, 3-4, 8-2
Heap 5-2

I

I/O Data Structures 8-2
IBM-PC 2-1, 5-5
icon.library 2-3
Iconic Tools 6-15
Identifier Scope 4-14
IEEE Floating Point Format 3-5
IEEE Floating Point 8-6
IEEE Math Routines 8-11
#if Command 4-10
In-line Code Generation 3-7
INCLUDE Assignment 1-7
include: E-2
Indirect Addressing 2-3
Initializer Expressions 4-14
Integer Constants 4-3
Integer Sizes 4-11
Internal Tables 8-1
intuition.library 2-3, 3-12
IntuitionBase D-8
iomode F-126
is F-128

K

Kernighan and Ritchie 3-4, 4-2, 4-18

L

Lattice dBCIII Library 3-13
Lattice MacLibrary 3-13
Lattice Screen Editor 3-1
Lattice Technical Support 3-4
LC Assignment 1-7
LC Command 9-1
lc C-8
lc1 C-19
lc2 C-21
lc: E-3
lcm.lib 3-4, 3-5
lcmffp.lib 3-5
lcmieee.lib 3-5
LCOMPACT 8-12
ldexp F-97
Level 0 Functions 8-8
Level 1 I/O Structures 8-2
Level 2 I/O 8-8
LIB Assignment 1-7
lib E-4abort...F-2
Library Changes 8-8
Library Closing 8-6
Library Direct Calls 3-6
Library Linking Order 3-5, 3-13
Library Offsets 3-7, 3-9
Library Opening 8-6
License Agreement 1-3
Line Control 4-8
Linked Overlays 6-8
Linking Math Libraries 8-10
Linking 3-3
Load Module Format 8-5
Load-and-Stay-Resident Code 5-12
Logical Name Assignments 1-8, 9-7

longjmp F-131
lsbrk F-133
lseek F-135

M

Macro References 4-9
main F-138
Mainframes 2-1
malloc F-18
Map Files 6-16
Math Libraries 3-4, 8-10
MathBase D-9
matherr F-141
mathffp.library 2-3
mathieeedoubbas.library 2-3
mathieeesingbas.library 2-3
MathTranBase D-11
mathtrans.library 2-3
Memory Allocation 5-2
Memory Expansion 1-2
Memory Fragmentation 5-4, 5-10
Memory Locations 2-1, 2-2
Memory Requirements 1-1
Memory Sections 5-3
memory F-144
MEMWATCH 5-14, 8-12
Minicomputers 2-1
Mixing Assembly Language Modules 5-11
mkdir F-147
_MNEED D-17
modf F-70
Motorola Fast Floating Point 3-5, 8-6
msflag D-10
_mstep D-18
Multiple Overlay Nodes 6-14

N

Nested Comments 4-2
nofill F-152
NTRYHUNK 6-11

O

Object File Format 8-4
Object Files 3-1
Object Modules 3-15
omd C-23
oml C-25, 8-12
onbreak F-148
onexit F-150
Open() Function 3-6, 8-8
OpenLibrary() Function 3-9
Operations, 68000 5-6
_OSERR D-19
Overlay Call Vector 6-19
Overlay Call Vectors 6-12
Overlay Level 6-19
Overlay Management 6-9
Overlay MAP File 6-11
Overlay Node 6-19
Overlay Ordinate 6-19
Overlay Reference 6-19
Overlay Sizes 6-8
Overlay Tree Structures 6-7
Overlay Tree 6-19
Overlay WITH Files 6-10
Overlays 6-7

P

PATH Command 1-6
PC-Relative Branches 5-12
perror F-154
Pointer Initialization Expressions 4-14

Pointer Portability 5-6
POPCLI 5-14, 8-12
Portability Classifications 9-3
poserr F-156
#Pragma Calls 3-7
Pre-processor Commands 4-9
Primary Expressions 4-4
printf Function 3-10
Processor, 6502 2-1
    68010 2-2, 5-4
    68020 2-2, 5-4
    8086 2-5, 5-4
    8088 5-4
    Z80 2-1
Processors, 8-bit 2-1
Program Control Flow 4-17
Program Sections 5-1
Programming Conventions 2-5
Programming Example 3-2, 3-14, 5-5, 5-14, 6-13, 9-4, 9-5
Prototype Header Files 3-6
Public Domain Software 3-15
putc F-89
putchar F-89
puts F-91

Q

qsort F-158
QUAD Assignment 1-7
quad: E-5

R

rand F-160
rbrk F-133
read F-162
READ.ME File 1-3
realloc F-18
Recursive Re-entry 4-21
Register Assignments 4-13

Register Saving 5-7
remove F-164
rename F-166
Return Values 9-5
Returning Aggregates 5-8
rewind F-104
rlsmem F-168
rlsml F-168
rmdir F-171
ROM Kernel Libraries 2-2
ROM Kernel Manual Errors 3-9
ROM Kernel Manual 1-1, 3-9
Root Directory 3-14
Root Node 6-19

S

S Directory 1-5
sbrk F-133
setbuf F-172
setjmp F-131
Short Integers 5-12
signal F-175
sizmem F-177
_SLASH D-22
Soft-Machine Architecture 2-2
Source Files 3-1
Special Hunks 6-18
srand F-160
Stack Overflows 8-9
Stack Size 5-2, 5-13
Statement Flow Constructs 4-17
Static Objects 4-12
stcarg F-178
stccpy F-180
stcd_i F-182
stcgfe F-184
stcgfn F-184
stcgfp F-184
stci_d F-188

stcis F-186
stcisn F-186
stclen F-217
stcpm F-191
stcpma F-191
Storage Class-Specifiers 4-5
Storage Classes 4-12
stpblk F-194
stpbrk F-196
stpchr F-198
stpchrn F-198
stpcpy F-180
stpdate F-200
stpsym F-202
stptime F-204
stptok F-206
strbpl F-208
strcat F-210
strchr F-198
strcmp F-212
strcpy F-180
strcspn F-186
strdup F-215
stricmp F-212
Strings 4-3
strins F-216
strlen F-217
strlwr F-218
strmfe F-219
strmfn F-221
strmfp F-223
strncat F-210
strncmp F-212
strncpy F-180
strnicmp F-212
strnset F-224
strpbrk F-196
strrchr F-198
strrev F-225
strset F-224
strsfn F-226

strspn F-186
strsrt F-229
strtok F-231
strtol F-233
Structure Declaration 4-6
Structure Pointers 8-1
Structures 4-8
strupr F-218
stspfp F-236
Stub Routines 3-6
System Configurations 1-2
system F-238

T

Task Priorities 2-2
tell F-135
Temperature Conversion Program 3-4
Temporaries 4-16
Text Buffers 4-9
Text() Function 3-8
Time-Slicing 2-1
time F-240
_timedata D-12
to F-241
Translated Mode 8-8
translator.library 2-3
trig F-243
Type Names 4-7
Type Punning 4-17
Type Specifiers 4-6
tzset F-246

U

Unary Operators 4-5
ungetc F-248
Union Declarations 4-6
Unions 4-8
UNIX 2-1, 4-1, 5-1, 5-6, 8-8, 9-2, 9-3

unlink F-164
utpack F-250
utunpk F-250

V

Version 1.1 Compatibilities 3-8
Void Keyword 4-2, 4-18

W

Workbench Environment 1-6
write F-162

X

XDEF Statement 5-8
XENIX 8-8, 9-3
XREF Statement 5-9

Z

Z80 Processor 2-1

Lattice

Lattice, Incorporated
Post Office Box 3072
Glen Ellyn, Illinois 60138
TWX 910-291-2190
Telex 532253

Lattice is a registered trademark of Lattice, Incorporated.

Printed in USA